Data Serving Systems in Cloud Computing Platforms

Sudipto Das

eXtreme Computing Group (XCG),

Microsoft Research (MSR)

Day 2 Morning Session

TRANSACTIONS ON CO-LOCATED DATA: A SURVEY OF SYSTEMS

Outline

- Production scale-out transaction systems
  - Cloud SQL Server (Microsoft)
  - Megastore (Google)
  - Espresso (LinkedIn)
- Research Prototypes
  - ElasTraS
  - G-Store
  - Hyder
  - Relational Cloud
  - Deuteronomy

CLOUD SQL SERVER (MICROSOFT)

Cloud SQL Server

[Bernstein et al., ICDE 2011]

- Transform SQL Server for Cloud Computing
- Small Data Sets
  - Use a single database
  - Same model as on premise SQL Server
- Large Data Sets and/or Massive Throughput
  - Partition data across many databases
  - Application code must be partition aware

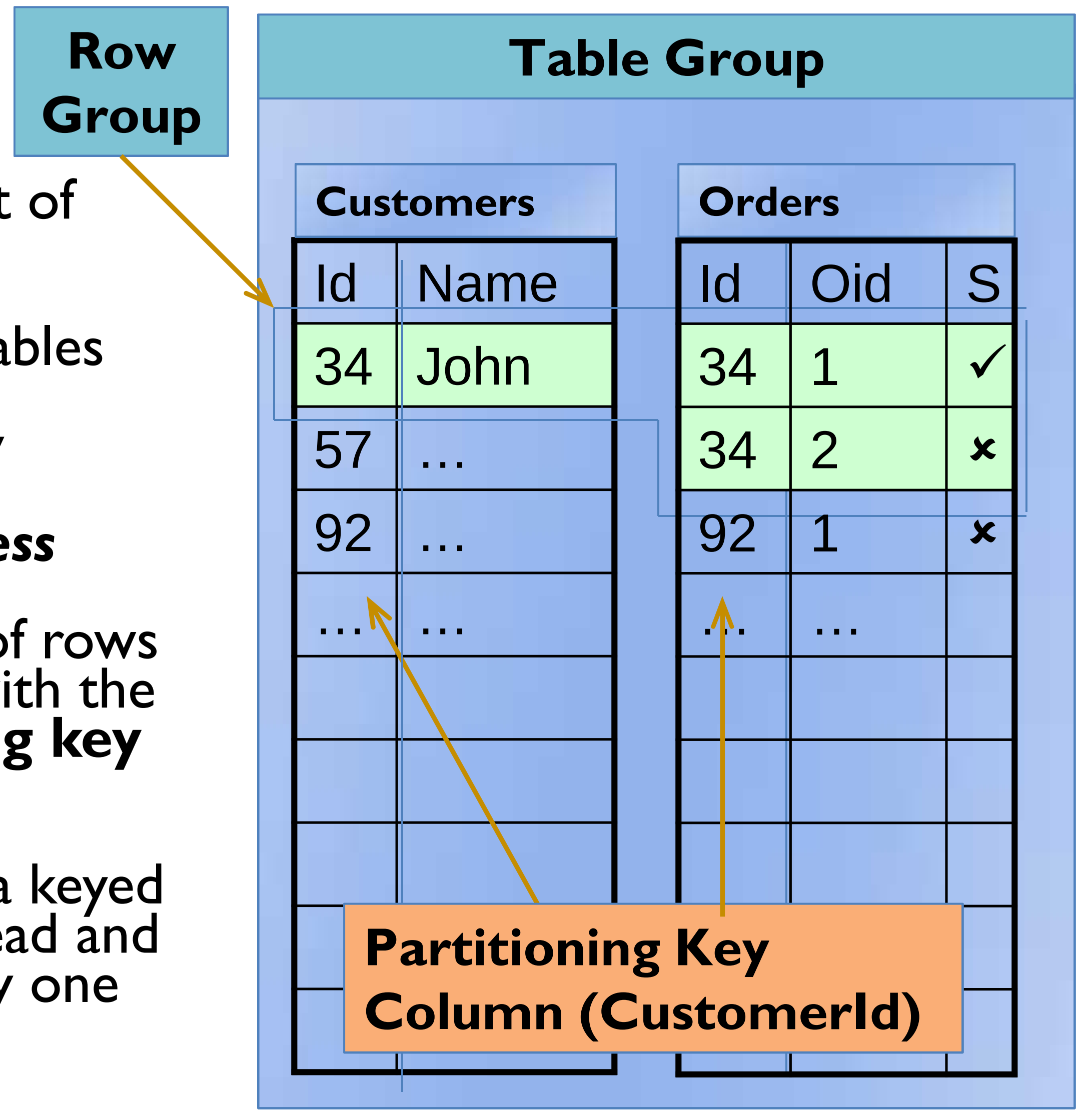
Design Philosophy

1. The application stores its data in multiple table groups, where each group fits in a single machine.
   - The application is responsible for scale out.
2. A keyed table group.
   - System responsible for scale out.

- No Two Phase Commit.

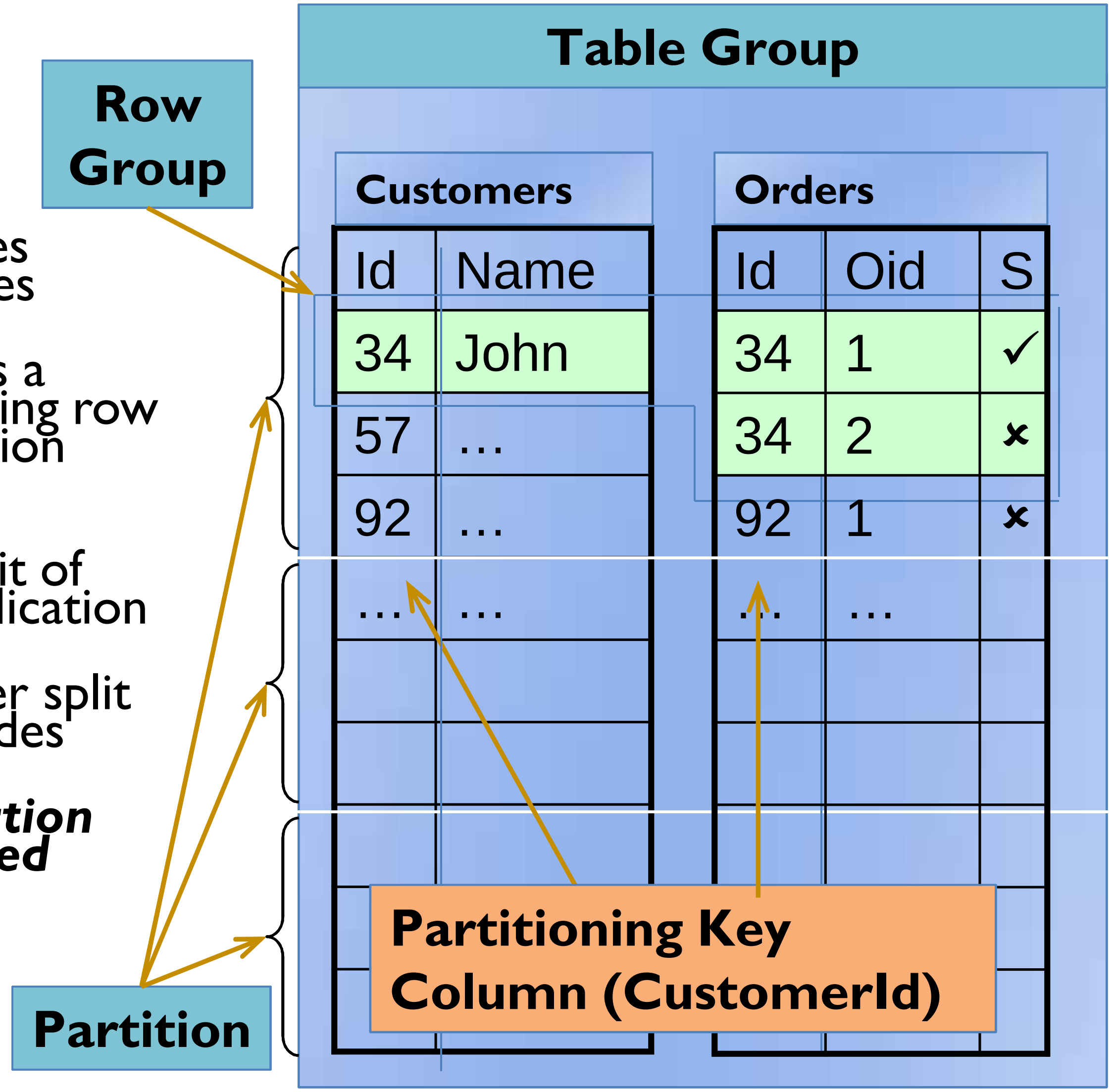
Logical Data Model

- ***Table group***: a set of tables
- If it is ***keyed***, all tables have the same **partitioning key**
- Or it can be ***keyless***
- **Row group**: set of rows in a table group with the same **partitioning key value**
- A transaction on a keyed table group can read and write rows of only one row group.

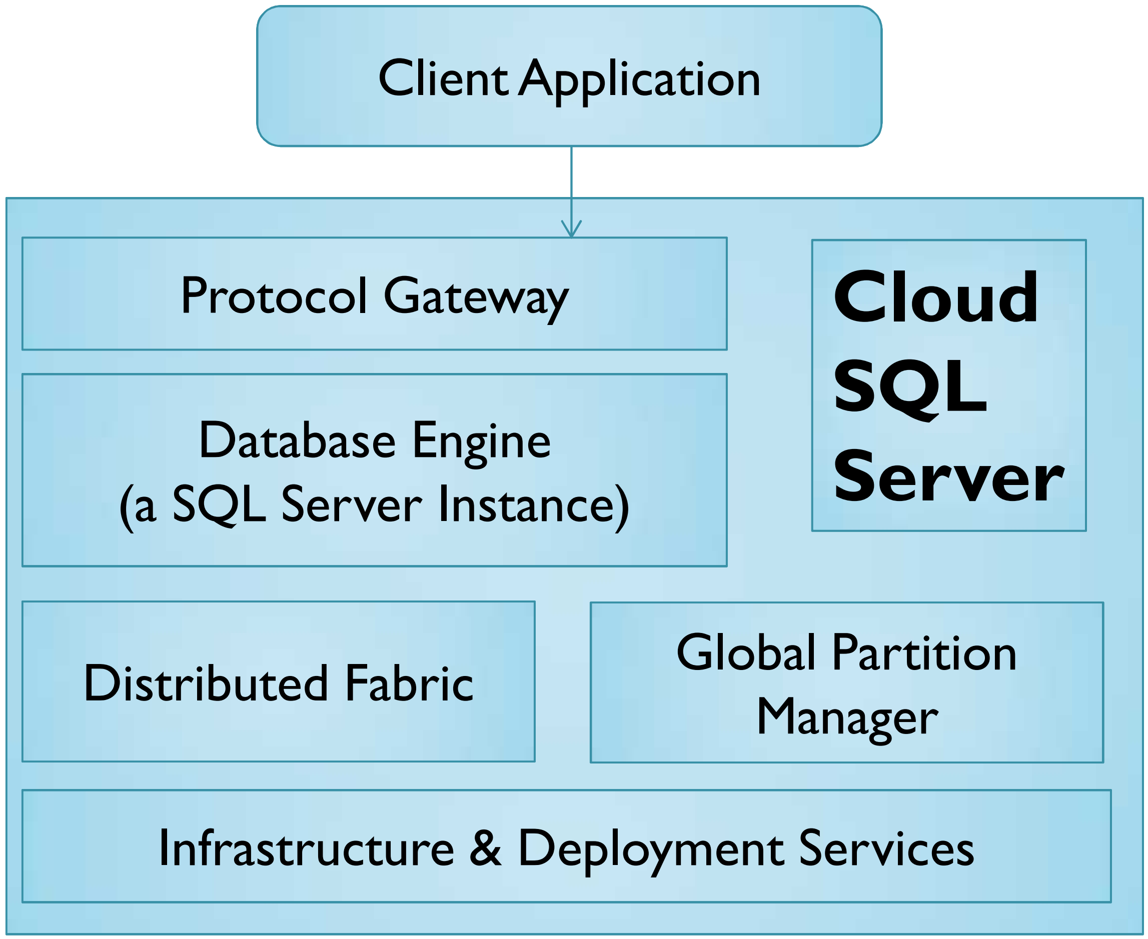


Physical Data Model

- Partition key values are split into ranges
- Each range defines a ***partition***, containing row groups with partition keys in the range
- Partition is the unit of distribution & replication
- A partition is never split across storage nodes
- ***Hence, a transaction is never distributed***

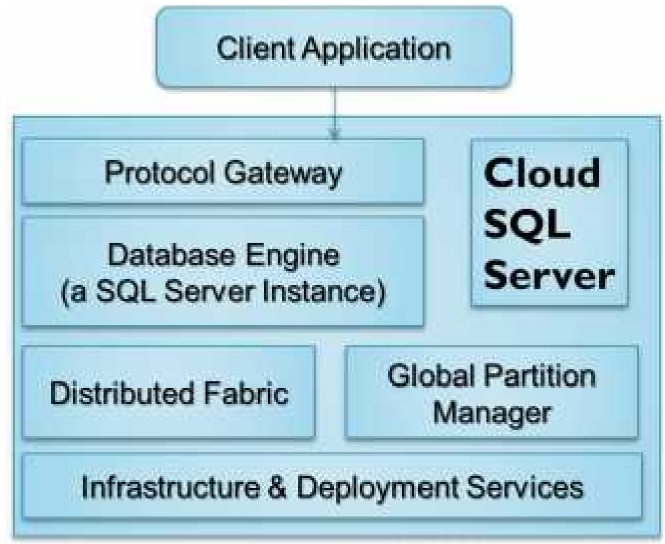


System Architecture



System Architecture

- Runs as one SQL Server instance
- I&D Services installs & upgrades images
- Fabric – DHT-based reliable sys management
  - Detects faults
- GPM – manages partition configuration
- Protocol gateway – manages sessions

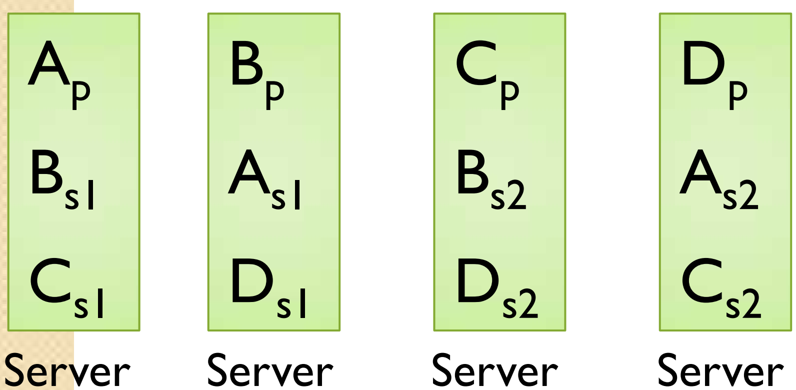


Upgrades Infrastructure and Deployment

- For each server S, Infrastructure & Deployment Services first checks with Global Partition Manager whether disabling S would cause a quorum loss
- If not, then it copies the image to S, disables S, installs the upgrade, and activates S
- Most upgrades have two phases: install & activate
  - Install everywhere before activating anywhere
  - Enables backing out if something goes wrong

Primary-copy Replication

- Each partition has multiple replicas
  - The global partition manager keeps track of this
- One is the ***primary***, which processes queries, updates, and DDL operations
- Secondary replicas are currently for fault tolerance
- Each storage node has a mix of primary and secondary partitions

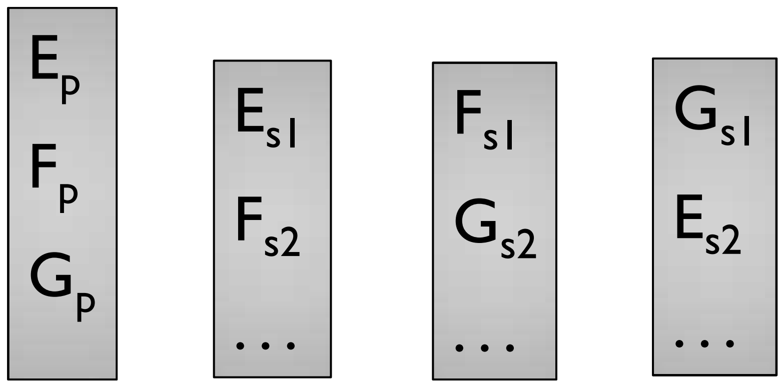


A_p = partition A, primary

A_{s1} = partition A, secondary 1

Replication & Load Balancing

- Table-group partitions are distributed independently
- Helps balance the load after a server failure



- If a server is overloaded, reassign a few primaries
- If a partition grows too large, split it
  - To avoid moving data, split primary and replicas
  - Reassign primary-hood to a split secondary

Replication – Normal Operation

- Primary eagerly sends update records for transaction T to each secondary
  - Contains key and after-image of payload
- Secondary buffers the updates for T
- If primary sends abort, secondaries discard T
- To commit T
  - Primary assigns commit sequence number (CSN) to T
  - Primary sends Commit to each secondary
  - Secondary runs a local transaction to install T's updates in CSN order and acks to Primary
  - After receiving acks from a quorum, primary commits T

Replication – Details

- Logs after-images, not operations or deltas
  - So replicas need not be identical
  - Avoids aligning disk allocation between replicas
- Logging index updates
  - Avoids pushing updates thru relational engine
  - Avoids a read to perform an update
- Use replication to distribute schema updates
  - Avoids special logic to synch data and schema updates
  - Job service sends schema updates to all partitions. So it only needs to track which *partitions* processed it, not which *replicas*

Replica Failure Handling

- If a secondary fails briefly, it gets the tail of the update log and catches up
- If a secondary S is down too long, GPM reassigns S to another server, which gets a copy of the primary
- If the primary is down, the GPM selects a leader to rebuild the configuration
  - If the leader can't reach a quorum of replicas, it declares "permanent quorum loss"
  - Else, it identifies the secondary with the latest state, which propagates updates to secondaries that need it
- In-flight transactions are resolved before the new primary starts new transactions

Replica Failure Handling (cont'd)

- GPM downshifts replica set to N-1.
  - If N=3 and a replica fails, downshifting avoids a quorum loss if a second failure occurs
- To determine latest state, each configuration of a partition has an epoch number
  - GPM increments epoch for each new configuration
  - Each commit record has an [epoch, CSN] pair
  - Latest commit is highest CSN within highest epoch
- GPM's database is replicated like other partitions
  - But if its primary fails, the fabric picks a new primary
  - Uses Paxos to ensure GPM epochs are totally ordered

Exchange Hosted Archive

- Archives messages and ensures compliance
  - Document retention policies
  - Document discovery for legal cases
  - Emergency email service when corporate email is down
- Partition key: tenant, time, content hash of message
- Uses many SQL Server features
  - non-clustered indexes
  - selection, aggregation, full-text queries
  - referential constraints
  - makes extensive use of stored procedures.
- Currently, 1000+ servers storing over a petabyte

SQL Azure – DB as a Service

- Access it like SQL Server
  - .NET Data Provider, Entity Framework, ODBC, PHP
  - Supports a large subset of SQL
  - Supports Integration Services, Analysis Services, and Reporting Services
  - Can use Sync Framework to sync with on-premise SQL Server or another SQL Azure DB
- First release uses keyless table groups
  - Enables rich SQL functionality
  - Since it's not partitioned, DB size limit is 50 GB.
- SQL Azure partitioning ("Federation") is coming

Highlights

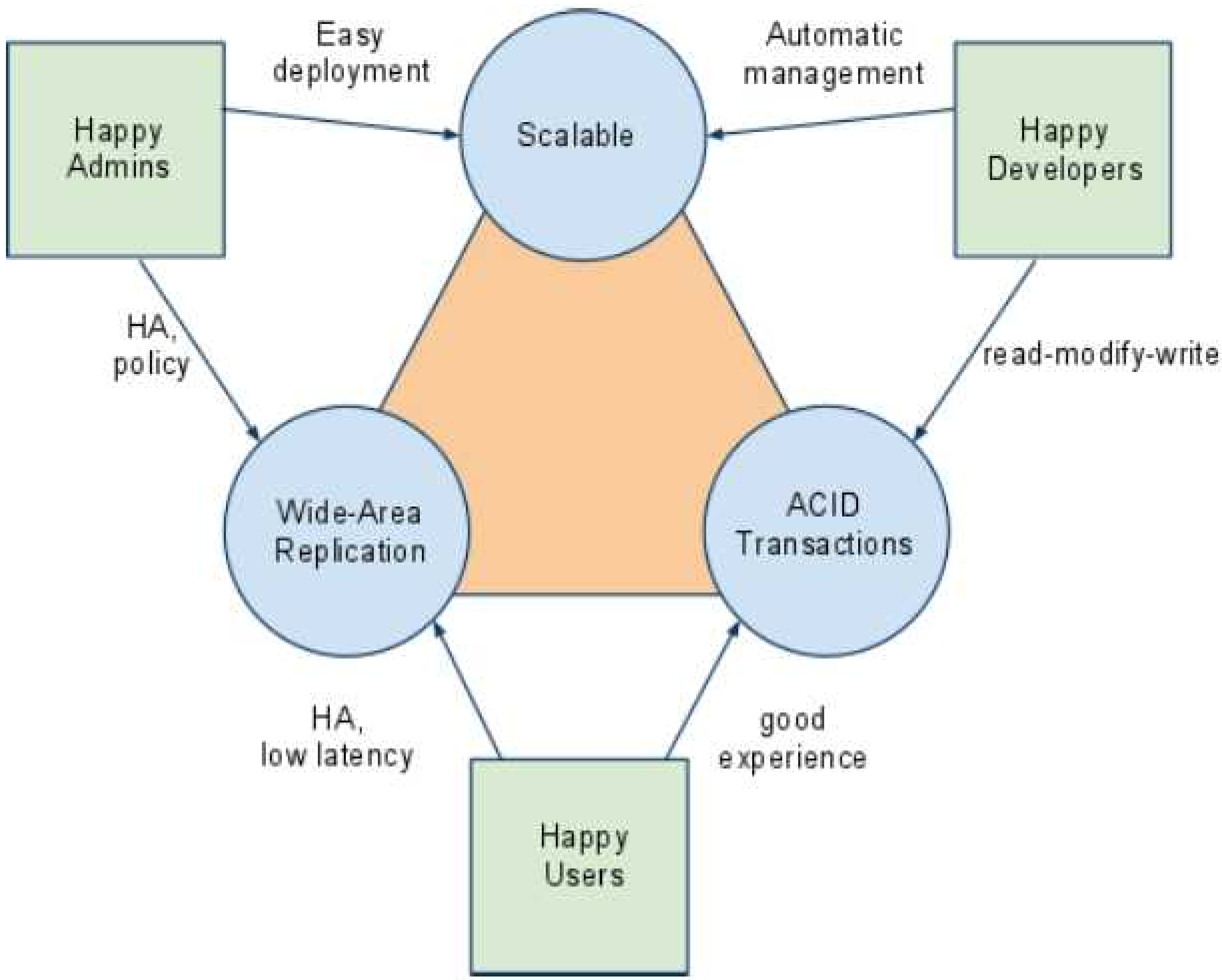
- Keyed partitions on one server.
- Simple one phase commit for replication
- Automated system management
  - Failure detection and recovery
  - Resource metering (for billing)

MEGASTORE (GOOGLE)

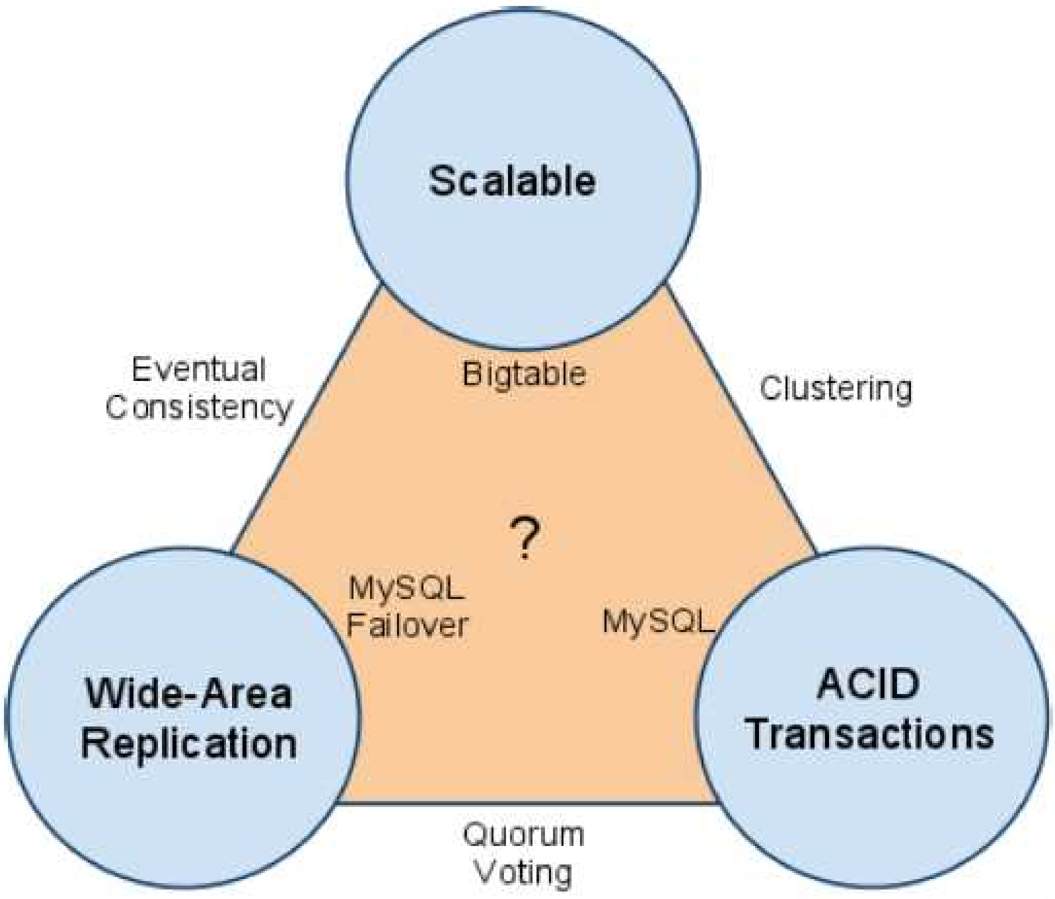
Megastore

- A billion Internet users
  - Small fraction is still huge
- Must please users
  - Bad press is expensive - never lose data
  - Support is expensive - minimize confusion
  - No unplanned downtime
  - No planned downtime
  - Low latency
  - Must also please developers, admins

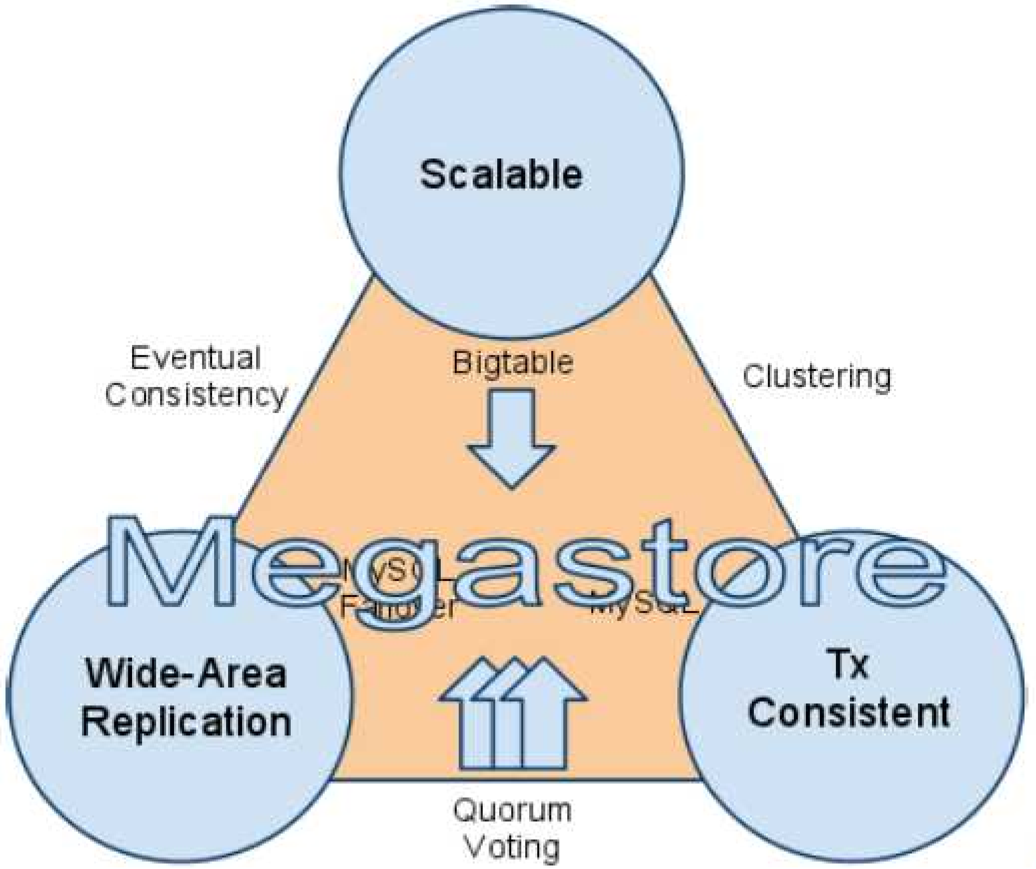
Making Everyone Happy



Technology Options



Technology Options



Megastore

[Baker et al., CIDR 2011]

- Transactional Layer built on top of Bigtable
- Entity Groups form the logical mini-database for consistent access
- Entity group: a hierarchical organization of keys
- Cheap transactions within entity groups
- Expensive or loosely consistent transactions across entity groups

Megastore

- The largest system deployed that use ***Paxos*** to replicate primary user data across datacenters on every write
- Key contributions
  - The design of a data model and storage system for rapid development of interactive applications
  - Optimized for low-latency operation across geographically distributed datacenters
  - Provides ACID semantics.

Toward Availability and Scale

- For availability
  - Synchronous, fault-tolerance log replicator
- For scale
  - Partitioned data into a vast space of small databases
  - Each with its own replicated log stored in a per-replica NoSQL datastore

Entity Groups

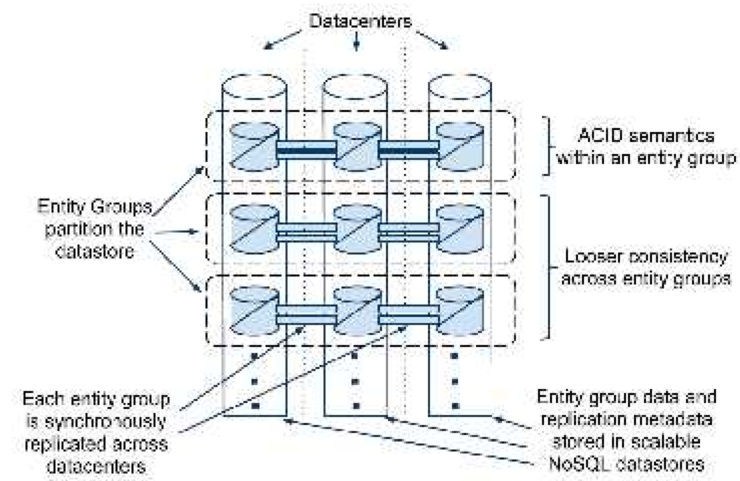
- Entity groups are sub-database (static partitioning)
  - Cheap transactions within Entity groups (common)
  - Expensive cross-entity group transactions (rare)

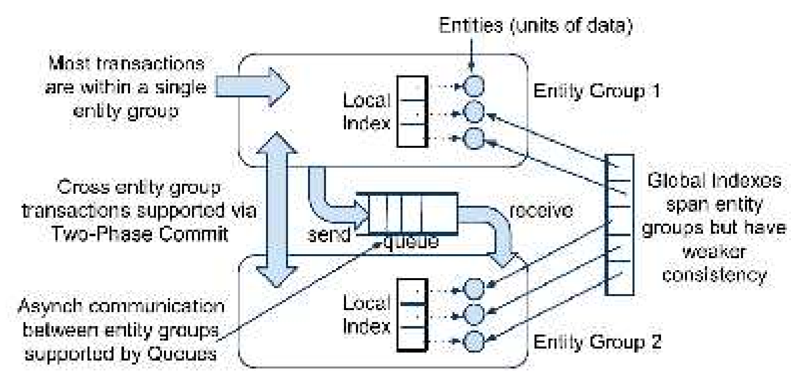
Replication

- Replicating data across hosts
  - High availability by overcoming sitefailures
  - ACID transactions are important
- Paxos algorithm
  - Proven, optimal, fault-tolerant consensus algorithm
    - No requirement for a distinguished master
    - Any node can initiate reads and writes of a write-ahead log
  - Replicated write-ahead logs

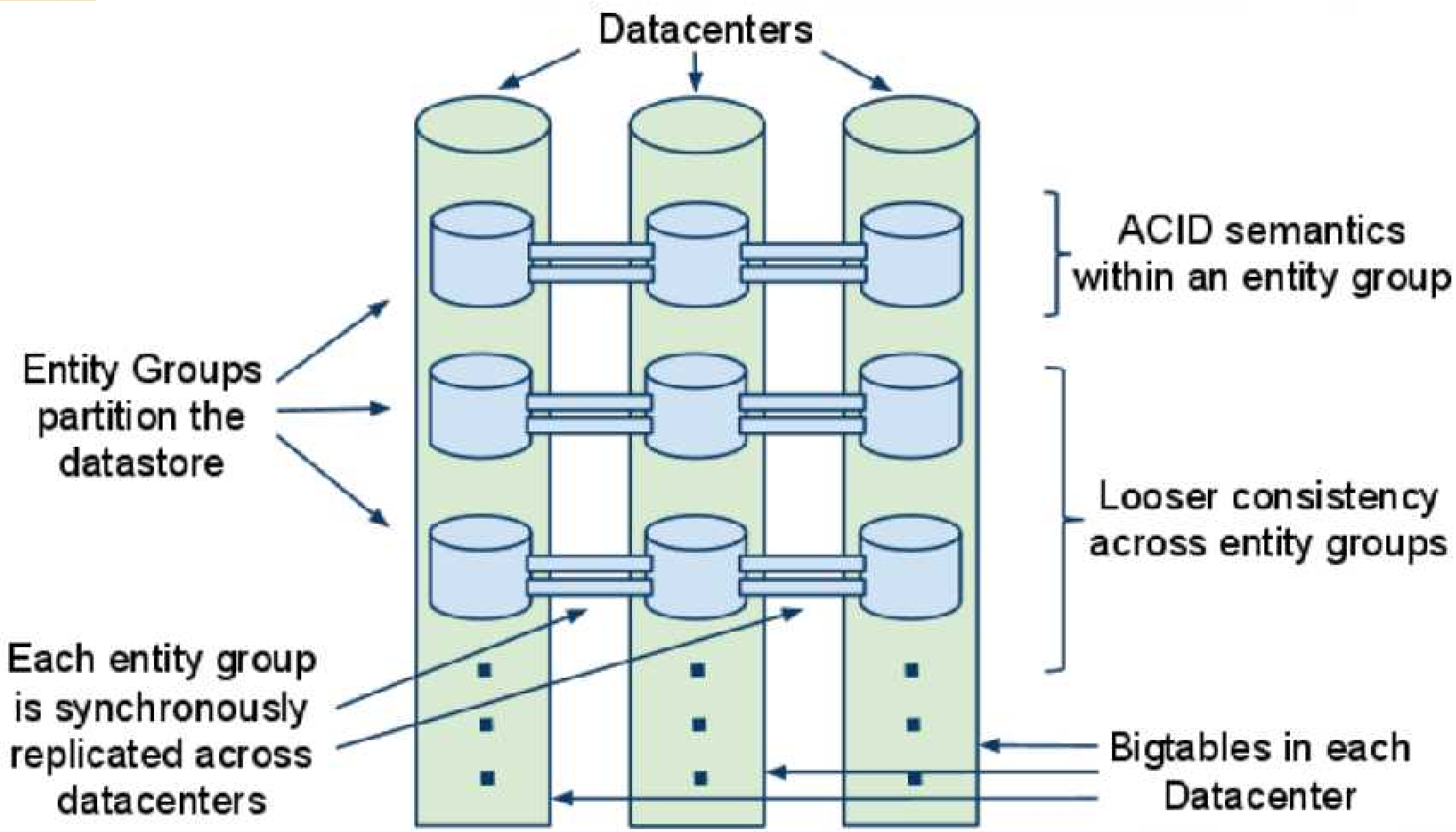
Partitioning and Locality

- For scale-up of replication
  - Entity groups
    - Data is stored in a scalable NoSQL datastore
    - Entities within entity group
      are mutated with single-phase
      ACID transactions
  - Operations
    - Cross entity group transactions supported via two-phase commits





Architecture



Entity Groups

- Examples of entity groups in applications
  - Email
    - Each email account forms a natural entity group
    - Operations within an account are transactional: user's send message is guaranteed to observe the change despite of fail-over to another replica
  - Blogs
    - User's profile is entity group
    - Operations such as creating a new blog rely on asynchronous messaging with two-phase commit
  - Maps
    - Dividing the globe into non-overlapping patches
    - Each patch can be an entity group

Transactions and Concurrency Control

- Each Megastore entity group functions as a mini-database with ACID semantics.
- A transaction writes its mutation into the entity group's write-ahead log, then the mutation is applied to data
- Recall, Bigtable can store multiple values in the same row/column pair with different timestamps.
- MVCC: multi-version concurrency control
- When mutations are applied, values are written at the timestamps of their transactions.

Concurrency Control

- Read consistency
  - Current: last committed value of a single entity, after all previous written values are applied.
  - Snapshot: reads single entity from the last fully applied transaction
  - Inconsistent reads: ignore the state of log and read the last values directly

Concurrency Control

- Write consistency
  - Always begins with a current read to determine the next available log
  - Commit operation
    - gathers mutations into a write-ahead log entry
    - assigns it a timestamp higher than any previous one
    - Appends to log using Paxos
  - Paxos uses optimistic concurrency : though multiple writers maybe attempting concurrently, only one wins.

Complete transaction lifecycle in Megastore

1. Read
   - Obtain the timestamp and log position of the last committed transaction
2. Application logic
   - Read from Bigtable and gather writes into a log entry
3. Commit
   - Use Paxos to achieve consensus for appending that entry to the log
4. Apply
   - Write mutations to the entities and indexes in Bigtable
5. Clean up
   - Delete data that is no longer required

Cross Entity group transactions

- Weak consistency. Using queues to provide asynchronous transactional messaging between entity groups, eg, if each calendar is an entity group, a single transaction can atomically send invitation queue messages to many distinct calendars. Not necessarily serializable.
- Strong Consistency, using two-phase commit: for atomic updates across entity groups. Discouraged.

Replication

- Single, consistent view of the data stored in its underlying replicas
- Characteristics
  - Reads and writes can be initiated from any replicas
  - ACID semantics are preserved regardless of what replica a client starts from
  - Replication is done per entity group
    - By synchronously replicating the group's transaction log to a quorum of replicas
  - Writes require one round of inter-data center communication
  - Reads observe last-acknowledged write and
    - After a write is observed, all future reads observe that write

Replication Options

- Master-based approach:
  - Limited flexibility for read and write operation
  - Master failover complicated
- Original Paxos:
  - Writes require 2 round trips (prepare and accept)
  - Reads require 1 round trip.
- Optimize Paxos: Megastore approach.

Megastore's Practical Paxos

- Fast Reads:
  - Current reads are executed locally on any replica.
  - Coordinator: A server in each data center, tracks the set of entity groups for which its replica has observed all writes. For these entity groups, replica serves local reads
  - Writes keep coordinator state consistent: If a write fails, the key is evicted from the coordinator state.

Megastore's Practical Paxos

- Fast Writes:
  - Single round trip writes using a notion of leaders.
  - Run Paxos for each log position.
  - The leader for each log position is a distinguished replica. Leader arbitrates which writer wins. First writer to the leader wins, and writes its value at all replicas, others use 2 phase Paxos.
  - Use closest replica as the leader for write, since most applications submit writes from the same region repeatedly.

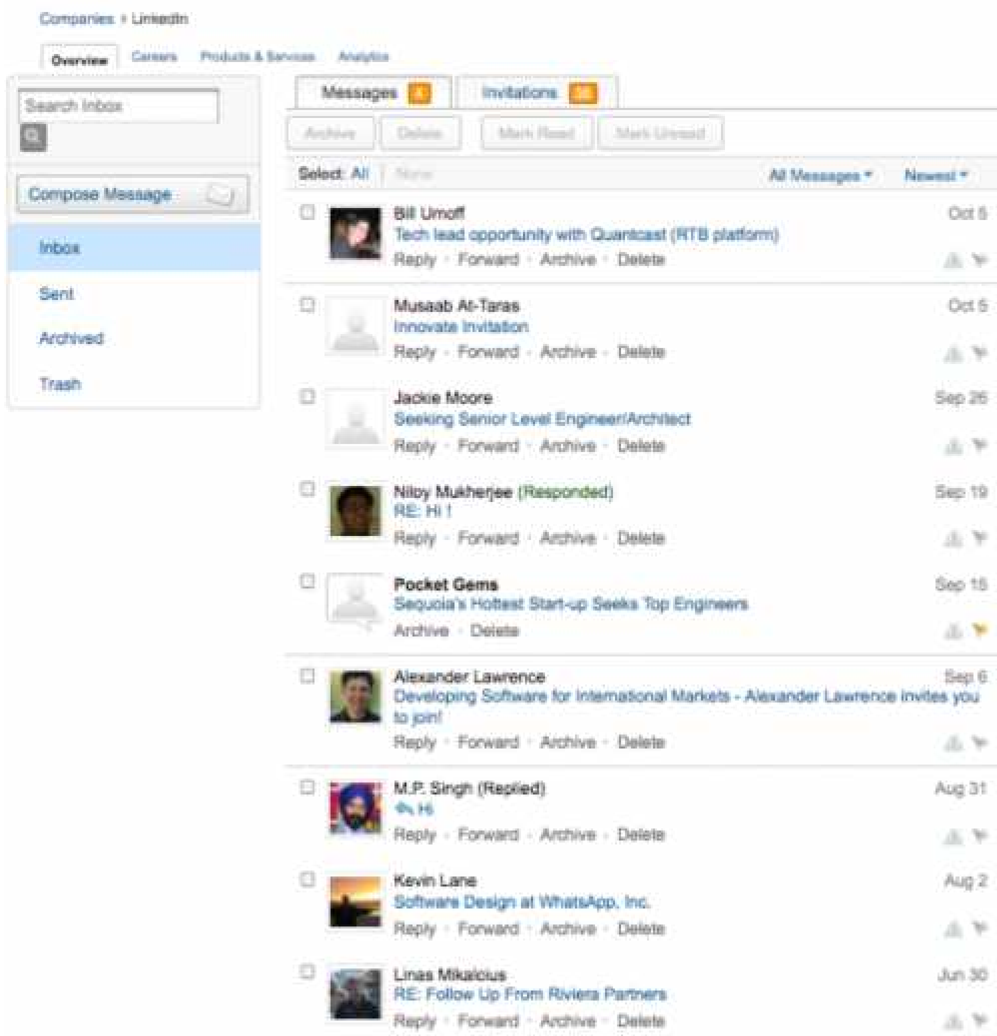
Highlights of Megastore

- Scale
  - Uses Bigtable within a datacenter
  - Easy to add Entity Groups (storage, throughput)
- ACID Transactions
  - Write-ahead log per Entity Group
  - 2PC or Queues between Entity Groups
- Wide-Area Replication
  - Paxos
  - Tweaks for optimal latency

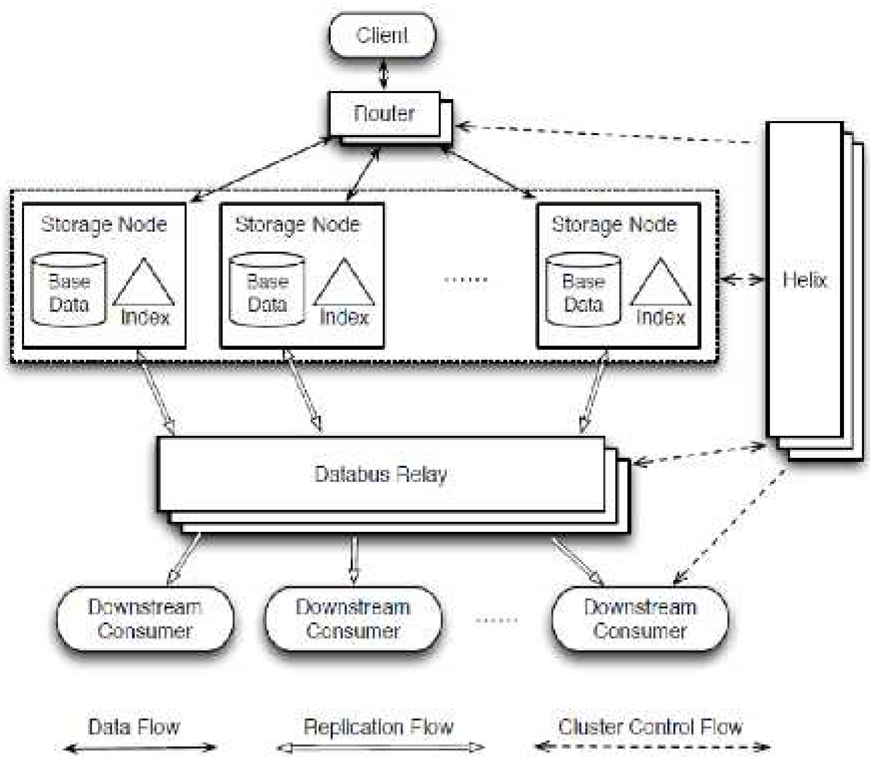
ESPRESSO: INDEXED TIMELINE-CONSISTENT DISTRIBUTED DATA STORE

Key Design points

- Hierarchical data model
  - **InMail**, Forums, Groups, Companies
  - Transaction support on related entities
- Produce native Change Data Capture stream
  - Timeline consistency
  - Read after write
- Rich functionality within a hierarchy
  - Local secondary indexes
  - Real-time updates to secondary indexes
  - Full-text search
  - ***On-the-fly schema evolution***
- ***Elasticity***
- Modular and pluggable
  - Off-the-shelf: MySQL, Lucene, Avro

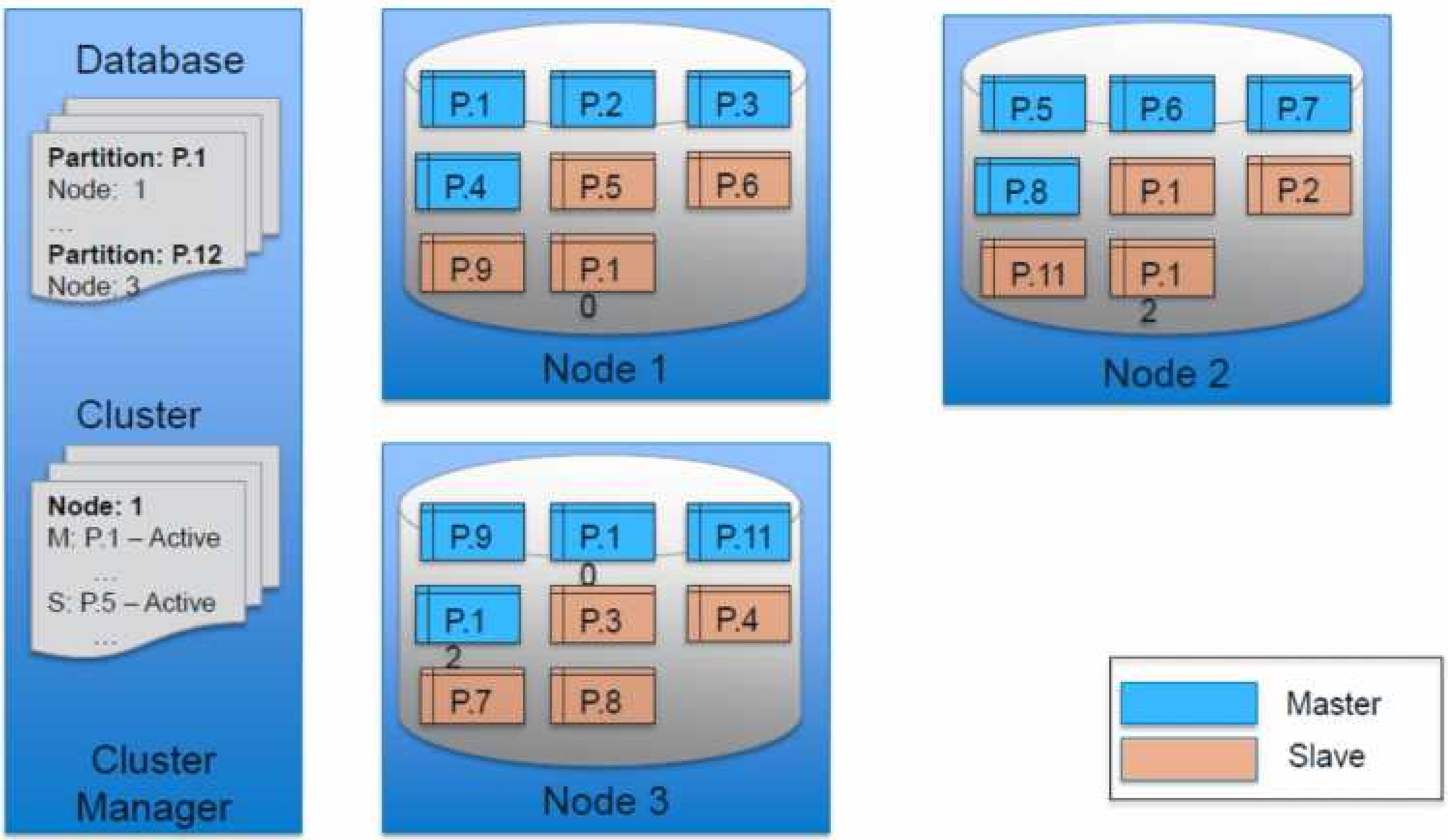


Architecture (10,000 ft)



- Major contributions
  - A novel generic distributed cluster management framework (helix)
  - A partition-aware change data capture pipeline
  - A high performance inverted index implementation

Architecture (1,000 ft)



Application View: Nested Entities

Mailbox Database

Message Metadata Table

MemberId	MsgId	Value Blob
bob	1	Invitation to join Linkedin
bob	2	Job opportunity
bob	3	Request for referral
tom	1	Invitation to join Linkedin
tom	2	Job opportunity

Mailbox Aggregates Table

MemberId	Value Blob
bob	unread:20, total:100
tom	unread: 2, total: 25

Message Details Table

MemberId	MsgId	Value Blob
bob	1	"Dear Bob,...."
bob	2	"Hello there,...."
bob	3	"Good morning, "
tom	1	"Hi Tom,..."
tom	2	"Interesting opportunity"

Partitioning

Hash or range partitioning of the ID space

Mailbox Database - Partition 1

Message Metadata Table

MemberId	MsgId	Value Blob
bob	1	Invitation to join Linkedin
bob	2	Job opportunity
bob	3	Request for referral

Mailbox Aggregates Table

MemberId	Value Blob
bob	unread:20, total:100

Message Details Table

MemberId	MsgId	Value Blob
bob	1	"Dear Bob,..."
bob	2	"Hello there,..."
bob	3	"Good morning,"

Mailbox Database - Partition 2

Message Metadata Table

MemberId	MsgId	Value Blob
tom	1	Invitation to join Linkedin
tom	2	Job opportunity

Mailbox Aggregates Table

MemberId	Value Blob
tom	unread: 2, total: 25

Message Details Table

MemberId	MsgId	Value Blob
tom	1	"Hi Tom..."
tom	2	"Interesting opportunity"

- ACID updates to data items within an entity
- Timeline-consistent CDC stream for updating independent entities

Espresso API (REST-ful)

- Read
  - Document lookup via keys or secondary indexes
  - Lookup by key, lookup by key prefix, lookup by a projection of fields
- Write
  - Insertion or full/parital update of a single document via a complete key
  - Auto increment of a key prefix
  - Transactional update of a document group
- Conditionals
  - Supported on both reads and writes
  - Used to implement equivalents of compare-and-swap
- Multi Operations
  - All reads and writes have their multi-counterparts for multi-operation transactions
- Change stream listener
  - API through DataBus

Storage Nodes

- Data stored and served by the individual storage nodes
  - Local transactional support for updates within a partition
  - Update base table and local indexes within a single transaction
- Replicas maintained by the change stream (using DataBus)
- Secondary indexes on the document groups (partitions)
  - Global secondary indexes implemented as derived tables (similar to that in PNUTS)

Replication and Consistency

- Primary-copy replication
  - Enhancements to MySQL's binary logging
  - Change events distributed using DataBus
  - Semi-synchronous (commit only after replication succeeds on at least one relay) or asynchronous replication
- Ordering of operations of primary similar to Lamport timestamps (system change number) appended to the node ID
- On master failure, slave promoted to master
  - Drain the change events from DataBus before serving requests
  - Might loose tail-of-log
- All replication within a single DC
  - Cross data center asynchronous replication for DR
  - DataBus to the rescue

Espresso Usage

- Company Pages
  - Over 2.6 million company pages
  - A company profile page may list one or more products
  - Products may have many recommendations
  - Hierarchy implemented as three tables with products listed under companies and recommendations listed under products
  - Read-heavy workload with 1000:1 ratio of reads to writes
- InMail
  - Message table: stores the raw messages
  - Mailbox table: summary view of the mailbox
  - Updates to a message table atomically updates the mailbox table
  - Write-heavy with 3:1 read to write ratio
- Unified Social Content Platform (USCP)
  - Shared platform that aggregates social activity across LinkedIn
  - Annotate a service's data with social gestures, such Likes, comments, and shares
    - E.g.: LinkedIn Today, Network Update Stream, LinkedIn Mobile

ELASTRAS TRANSACTION MANAGEMENT (UCSB)

Elastic Transaction Management

[Das et al., ElasTraS, HotCloud 2009, TODS 2013]

- **On-demand Scalability → Elasticity**
- Database viewed as a **collection** of **partitions**
- Suitable for:
  - Large single tenant database
    - Database partitioned at the schema level
  - Multitenant databases
    - Large number of small databases
    - Each partition is a self contained database

Elastic Scalability

- **Decouple ownership from storage**
  - Working sets fit in cache
    - Negligible performance impact
  - Simplifies transaction management
    - No need to handle replication in TM layer
  - Low cost migration
    - lightweight elasticity
- **Limit interactions to a single node**
  - Efficient (non-distributed) transaction execution
  - Loose synchronization between nodes
    - linear scalability

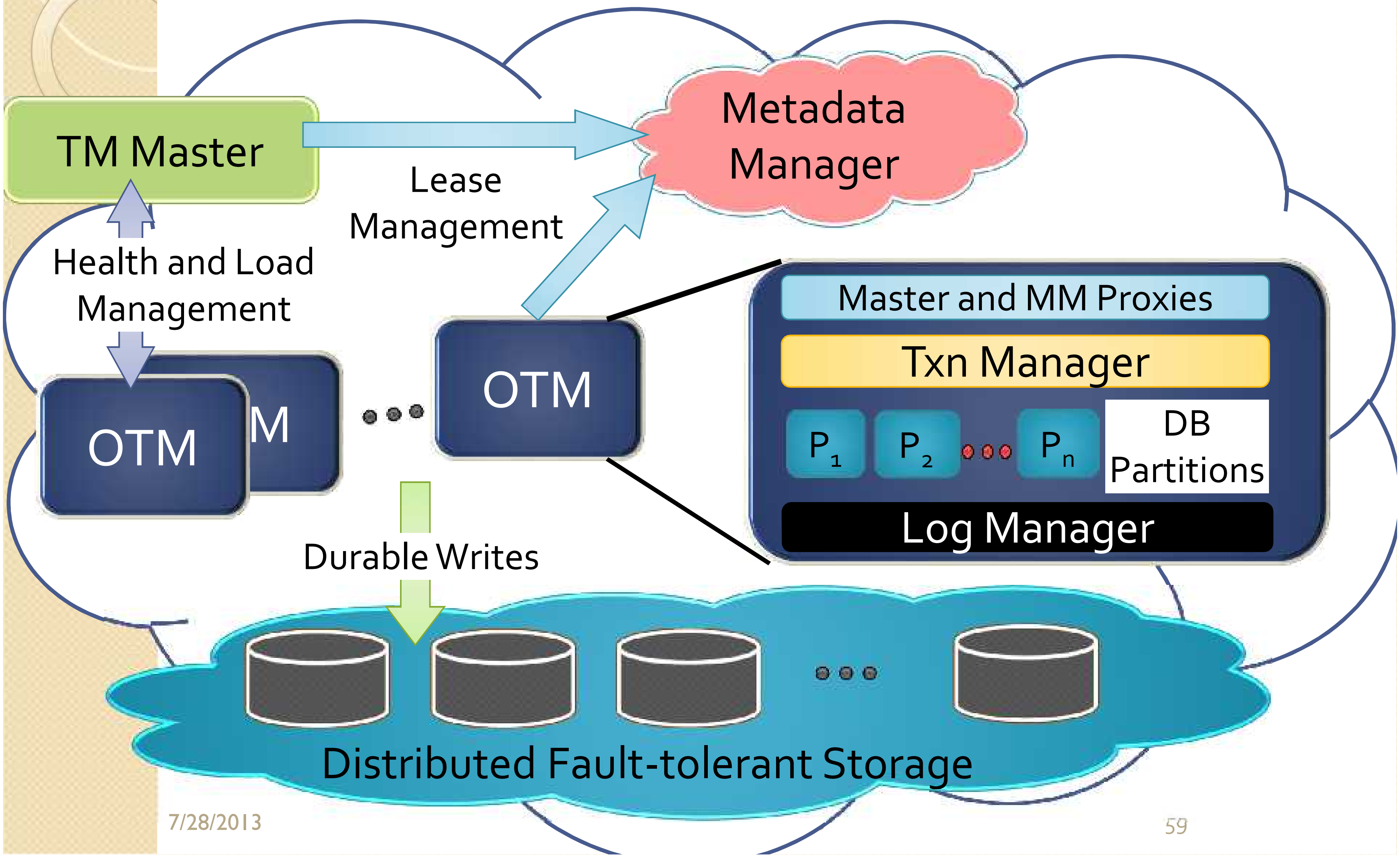
Design Rational

- **Separate System and Application state**
  - System State
    - Partition to server Mapping
    - Lease information
  - Application State
    - Data served by OTMs
- **Limited distributed synchronization**
  - Loose coupling between OTMs, TM Master, and Metadata Manager

ElasTraS

- Elastic to deal with workload changes
- Dynamic Load balancing of partitions
- Autonomic recovery from node failures
- Transactional access to database partitions

Overview of ElasTraS Architecture



Effective Resource Sharing

- Multiple database partitions hosted within the same database process
  - Shared process multitenancy
  - Allows better consolidation
  - Use conventional RDBMS engines
- Independent transaction and data managers
  - Good performance isolation

Transaction Management Layer

- **Concurrency Control**
  - OTMs execute transactions on partitions
  - Optimistic Concurrency Control
- **Recovery**
  - Transaction's updates logged before commit
  - REDO-only recovery after a failure
- **Storage and Cache Management**
  - Append only storage layout
  - Separate Read and Write Caches
  - Similar to Bigtable storage layout

Management and Control Layer

- **System metadata is critical**
  - Consensus based replication for strong consistency and high availability (based on Paxos)
  - Zookeeper in our implementation
- **TM Master monitors the system**
  - Detect failures
  - Coordinate recovery
  - Loose synchronization between nodes using leases

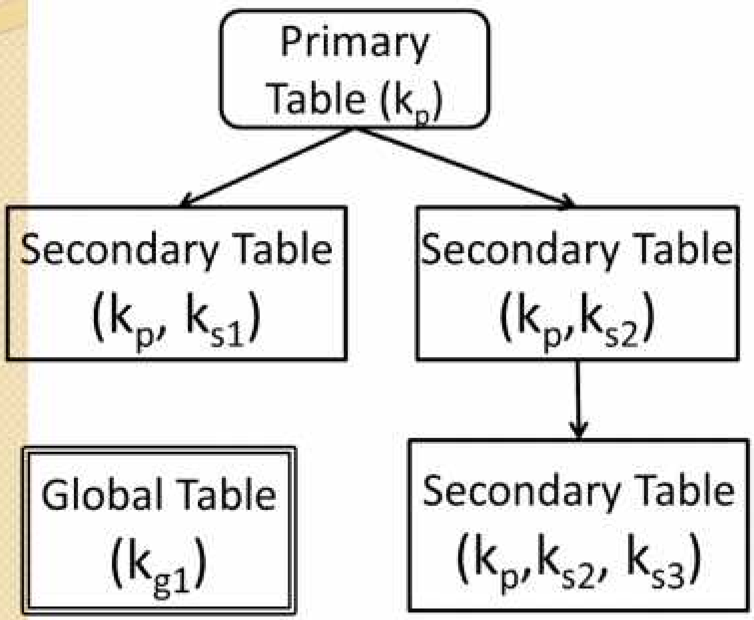
Elasticity and Load Balancing

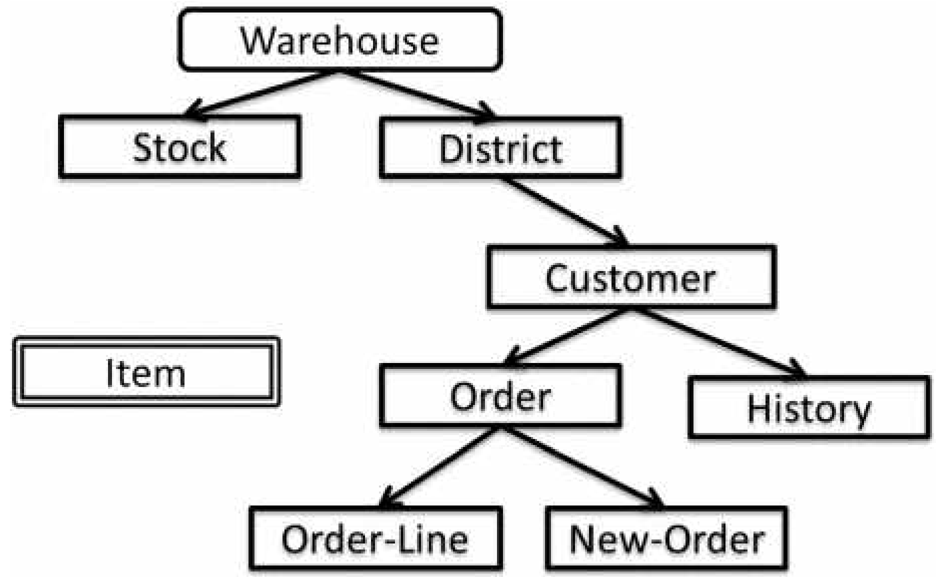
- TM Master monitors performance
  - Periodically obtain load and resource usage information
- Model the system's performance
- Determine
  - *Which* partition to migrate
  - *Where* to migrate
  - *When* to migrate
- Live Database Migration for elastic load balancing

Schema Level Partitioning

- Partition based on schema, not individual tables
- Cluster frequently accessed data items in a partition
- Leverage Access patterns in the workloads
  - **Tree schema**

Tree Schema


Primary Table (k_p)
Secondary Table (k_p, k_{s1})
Secondary Table (k_p,k_{s2})
Global Table (k_{g1})
Secondary Table (k_p,k_{s2}, k_{s3})

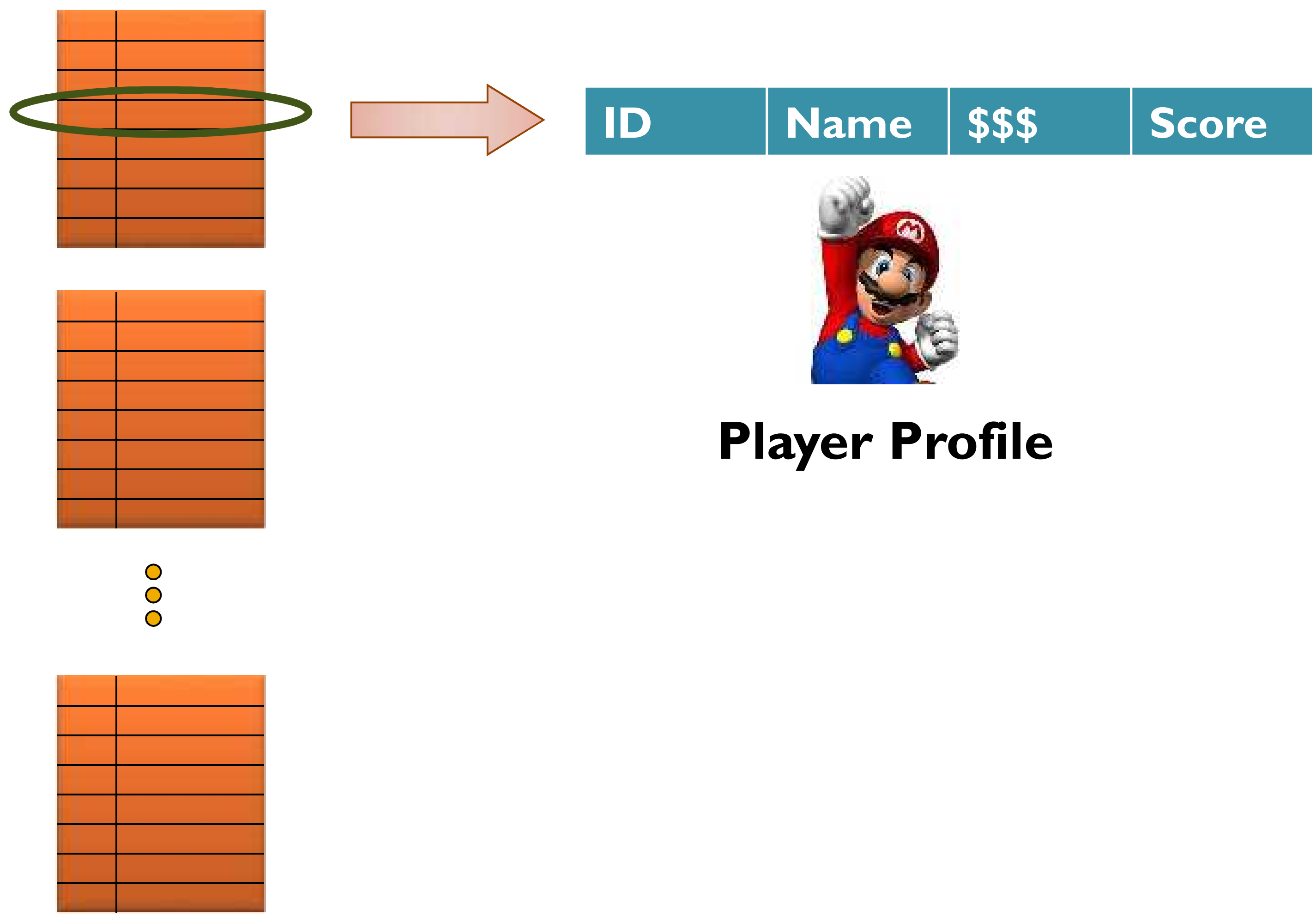

Warehouse
Stock
District
Customer
Item
Order
History
Order-Line
New-Order

DYNAMIC PARTITIONING: G-STORE (UCSB)

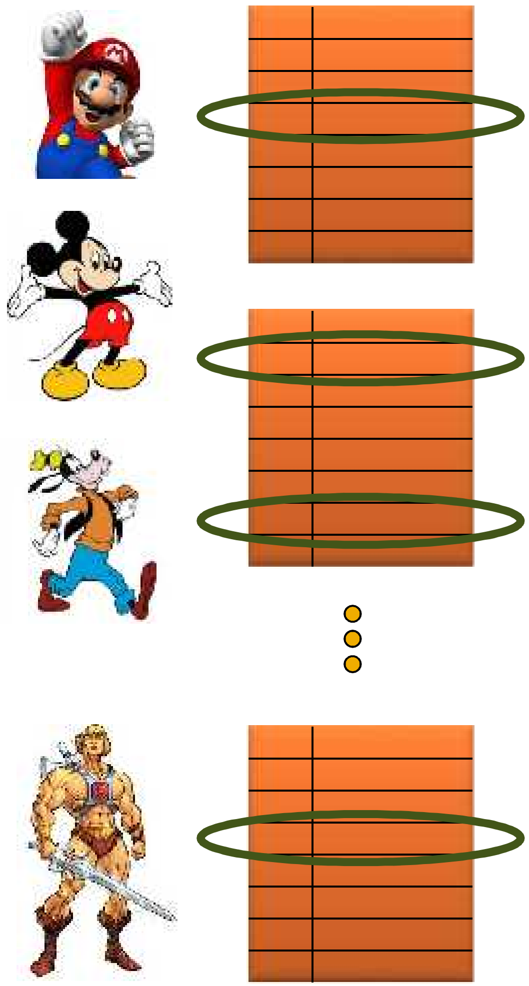
Dynamically formed partitions

- Access patterns evolve, often rapidly
  - **Online multi-player gaming** applications
  - **Collaboration** based applications
  - **Scientific computing** applications
- Not amenable to static partitioning
  - Transactions access multiple partitions
  - **Large numbers of distributed transactions**
- How to efficiently execute transactions while **avoiding** distributed transactions?
  - G-Store [Das et al., SoCC 2010] presents a solution

Online Multi-player Games

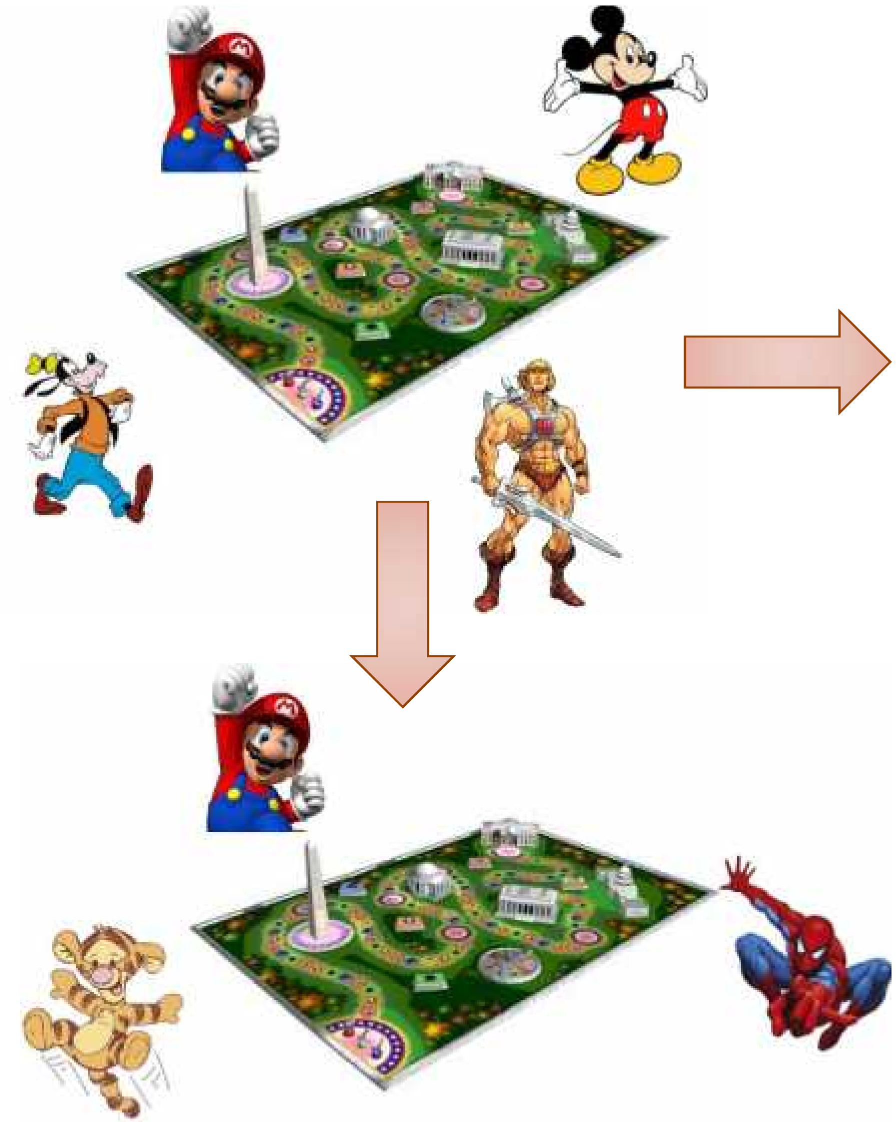


Online Multi-player Games



Execute transactions on player profiles while the **game is in progress**

Online Multi-player Games



Partitions/groups are **dynamic**

Online Multi-player Games

Hundreds of thousands of **concurrent** groups

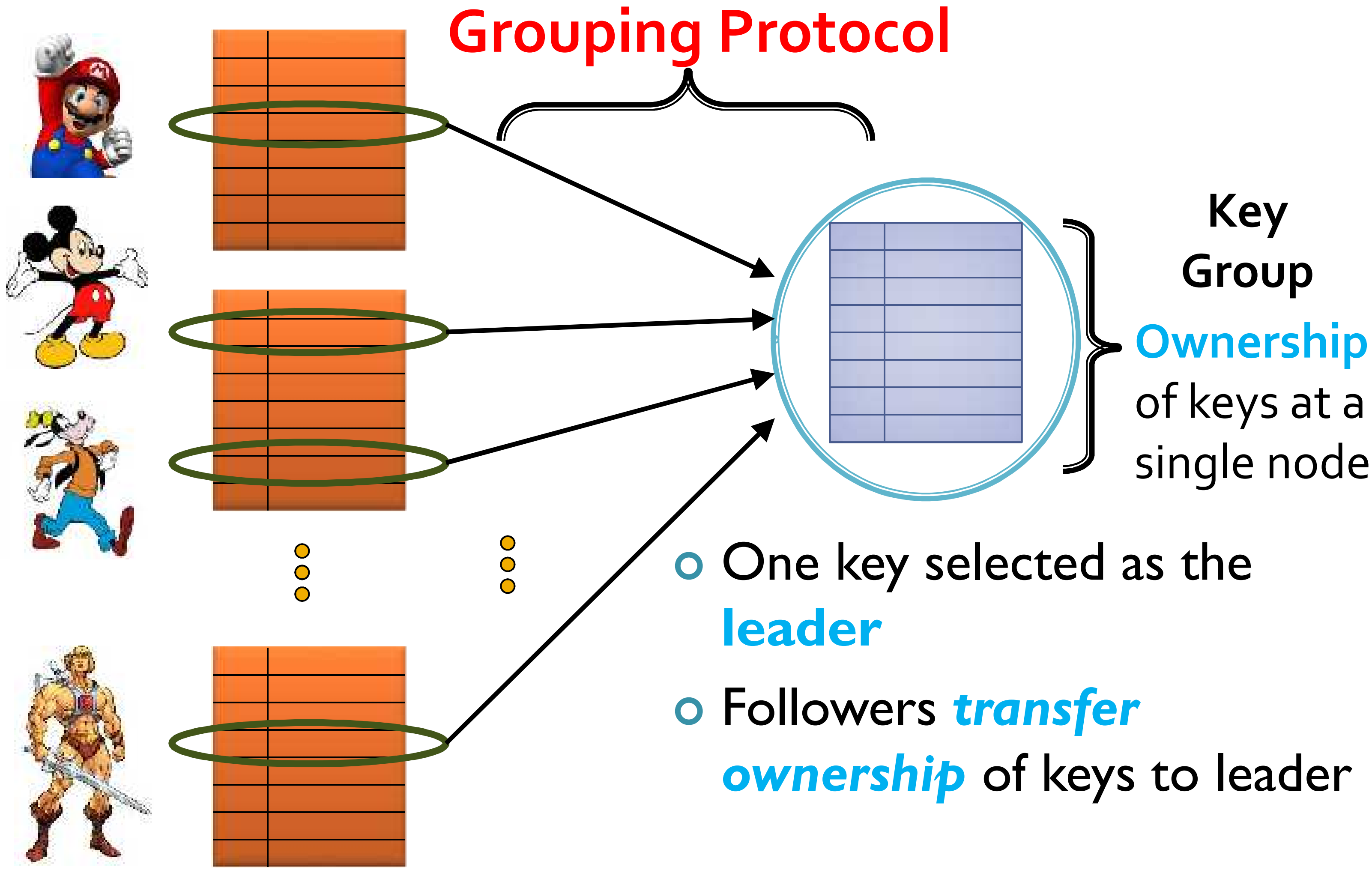
G-Store

[Das et al., SoCC 2010]

- ***Transactional*** access to a ***group of data items*** formed ***on-demand***
  - **Dynamically formed** database partitions
- ***Challenge:*** Avoid distributed transactions!
- ***Key Group Abstraction***
  - Groups are ***small***
  - Groups have ***non-trivial lifetime***
  - Groups are ***dynamic*** and ***on-demand***

Transactions on Groups

Without distributed transactions

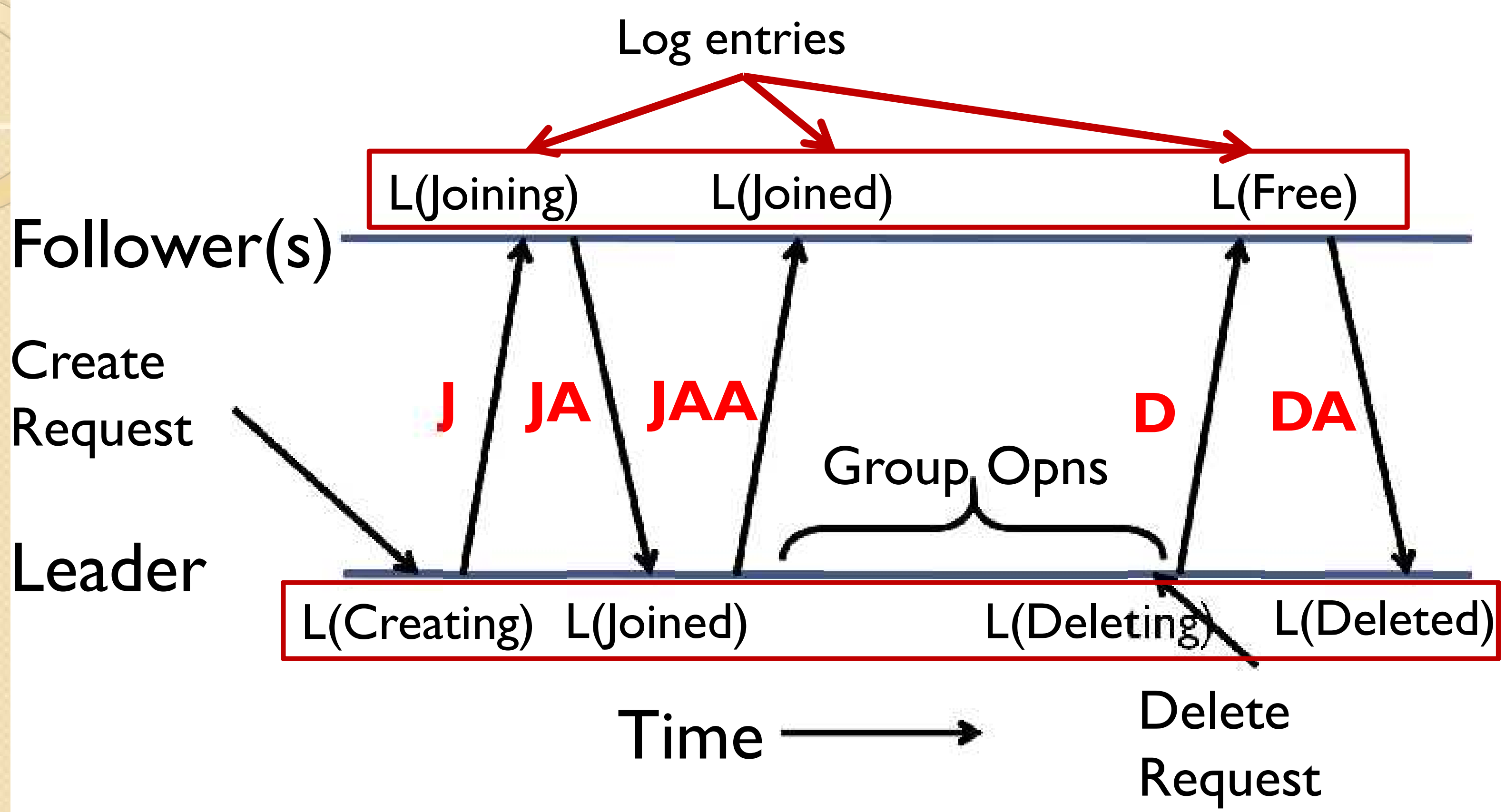


- One key selected as the **leader**
- Followers ***transfer ownership*** of keys to leader

Why is group formation hard?

- **Guarantee** the **contract** between **leaders** and **followers** in the presence of:
  - Leader and follower **failures**
  - Lost, duplicated, or re-ordered messages
  - Dynamics of the underlying system
- How to ensure **efficient** and **ACID** execution of transactions?

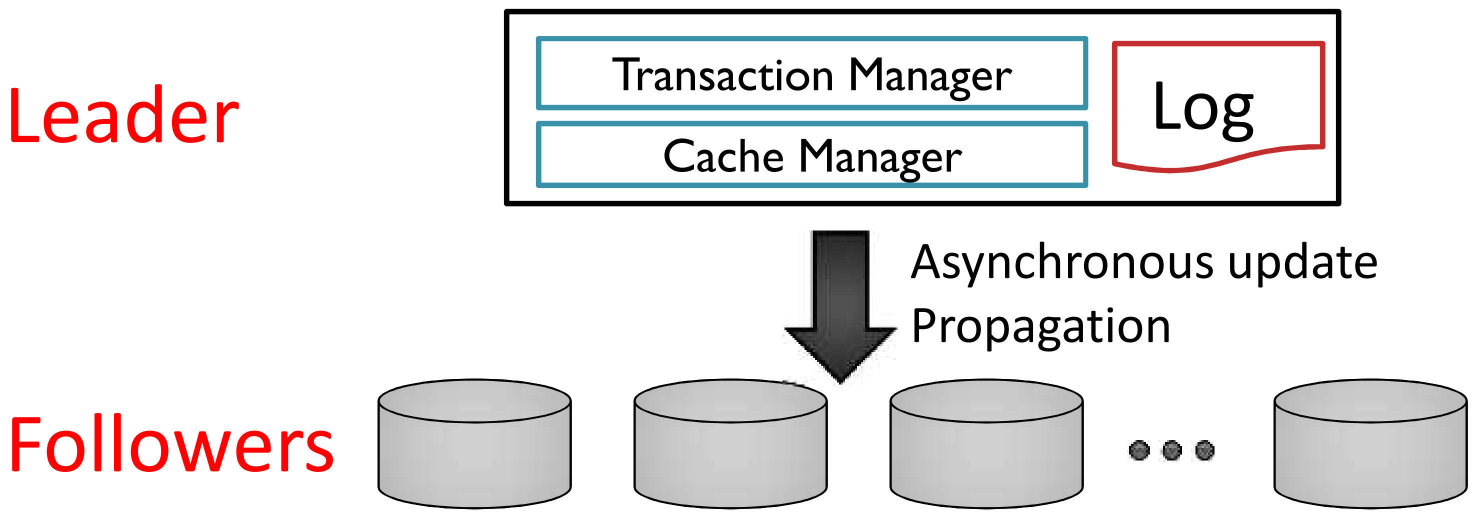
Grouping protocol



- Handshake between **leader** and **follower(s)**
  - Conceptually akin to **"locking"**

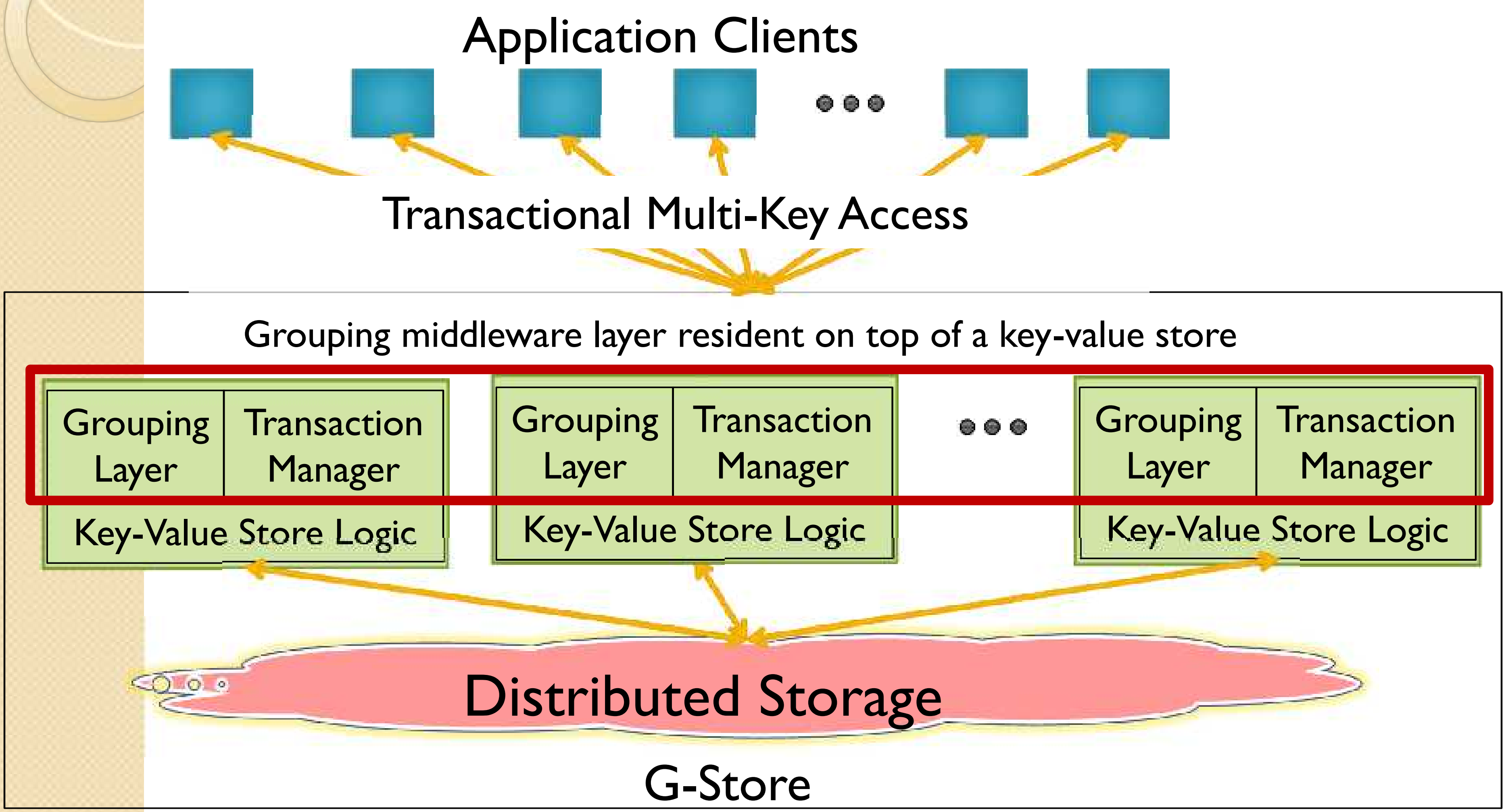
Efficient transaction processing

- How does the leader execute transactions?
  - **Caches data** for group members ➔ underlying data store equivalent to a disk
  - **Transaction logging** for durability
  - Cache **asynchronously flushed** to propagate updates
  - **Guaranteed update propagation**



Prototype: G-Store

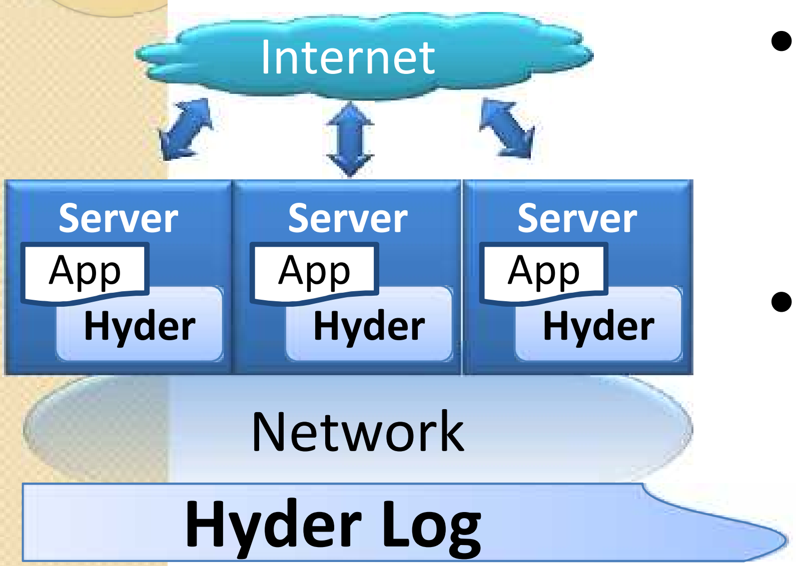
An implementation over Key-value stores



HYDER – A TRANSACTIONAL RECORD MANAGER FOR SHARED FLASH

Hyder: The Big Picture

Goal: Enable scale-out without partitioning DB or app



- Store the whole DB in flash
  - which is accessible to all servers
  - via a fast data center network
- Main architectural features
  - Uses a log-structured DB in flash
  - Broadcast log to all servers
  - Roll forward log on all servers
  - Optimistic concurrency control

- There's no cross-talk between servers
  - Hence, Hyder scales-out without partitioning

What is Hyder?

A software stack for transactional record management

- Stores [key, value] pairs, which are accessed within transactions

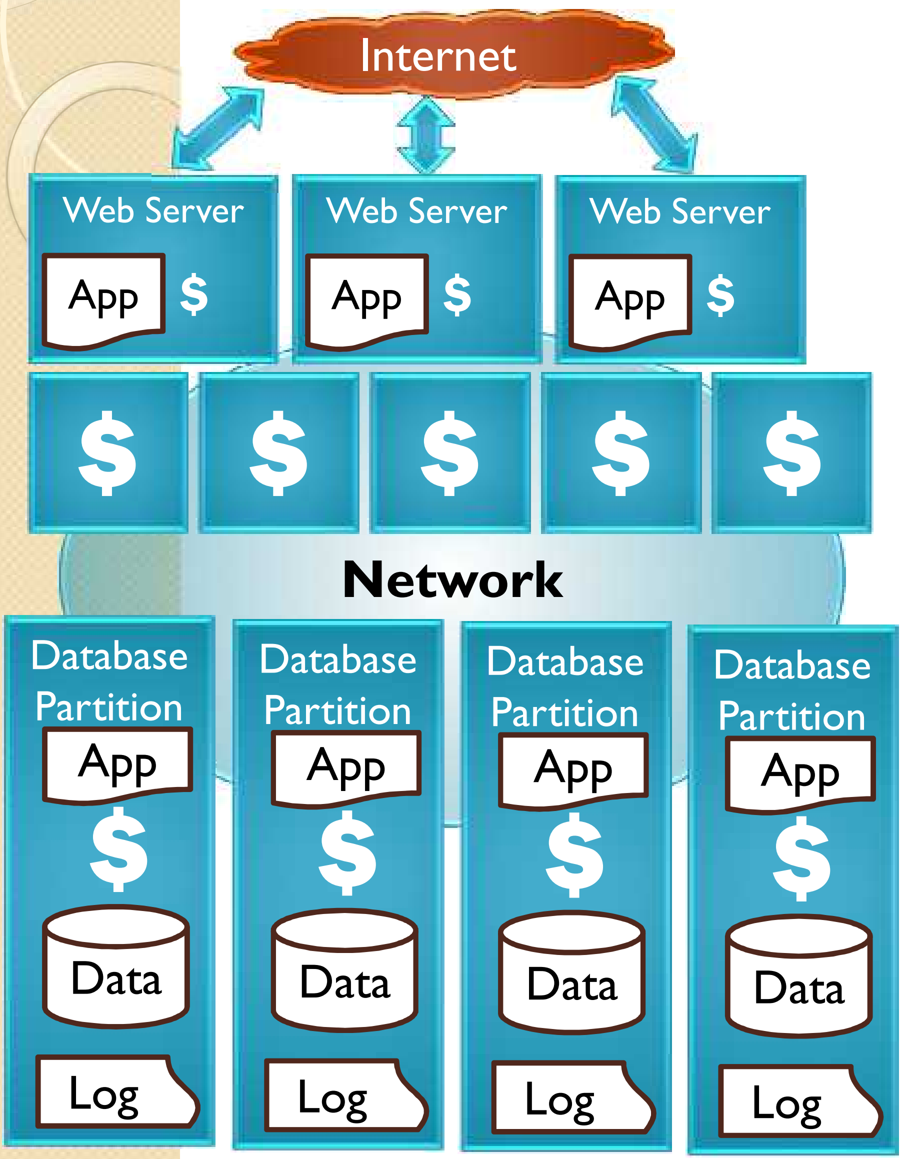
Functionality

- Record operations:
  - Insert, Delete, Update, Get where field = X; Get next
- Transactions: Start, Commit, Abort

Why build another one?

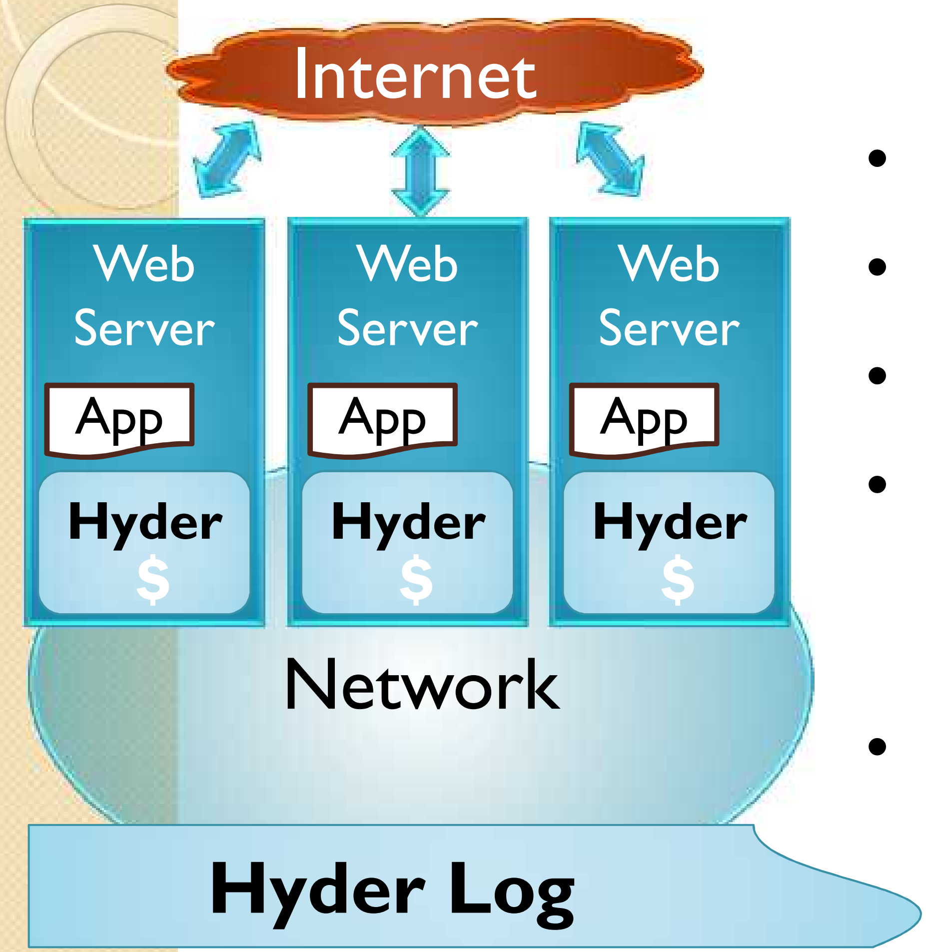
- Exploit flash memory and high-speed networks
  to simplify scaling out large-scale web services

Scaling Out with Partitioning



- Database is partitioned across multiple servers
- Each query is sent to the appropriate partition(s)
- For scalability, avoid distributed transactions
- Cross partition consistency is enforced in the application
- Hard to provision servers and distribute load evenly

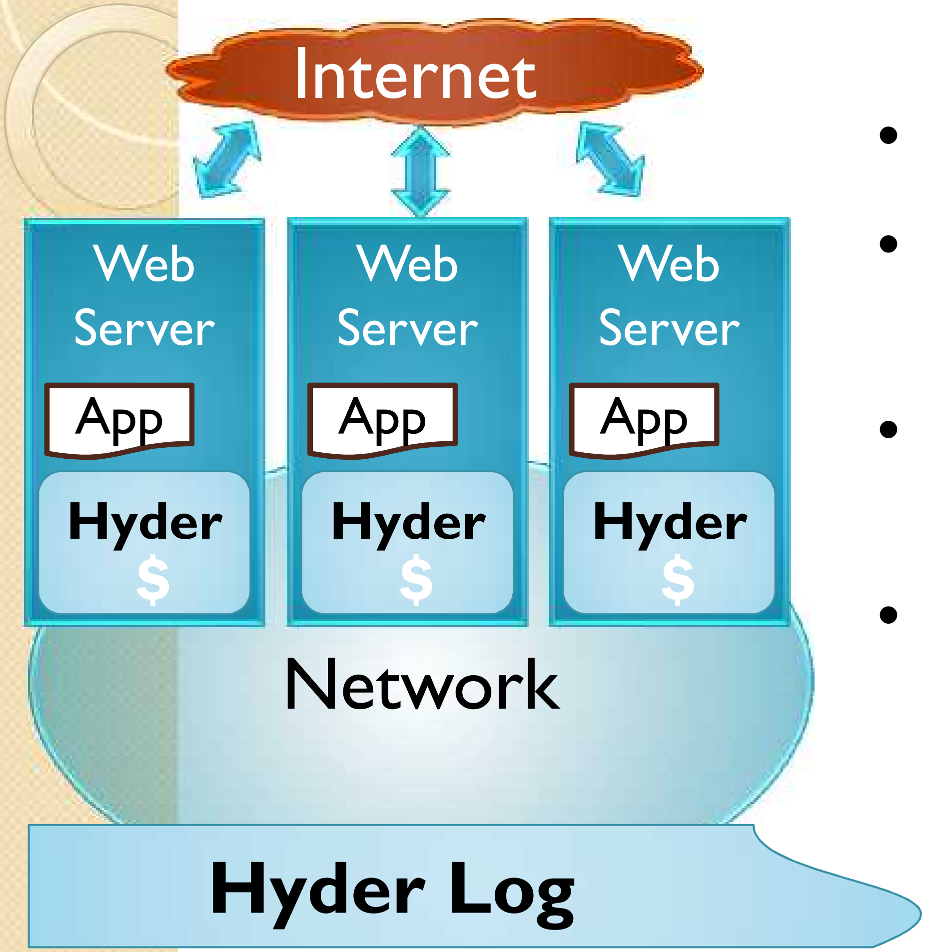
Hyder Scales Out Without Partitioning



- In Hyder, the log is the database
- All servers can access the log
- No partitioning is required
- Database is multi-versioned, so server caches are trivially coherent
- Hence, can parallelize a query with consistency across servers

And servers can fetch pages from the log or from neighboring servers' caches

Hyder Runs in the Application Process

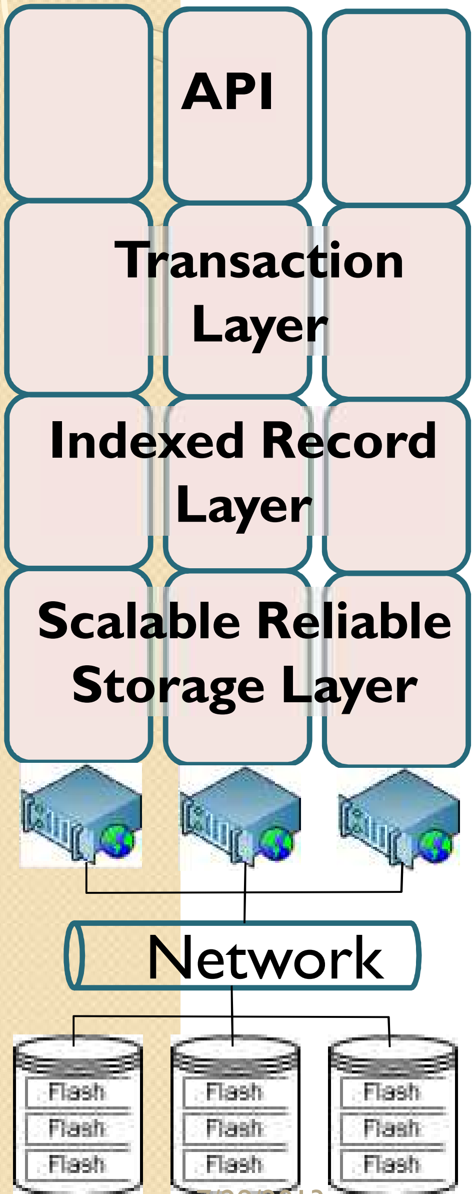


- No distributed programming
- No distributed caches for the app to keep consistent
- Avoids the expense of RPC's to a database server
- Simple high performance programming model

Enabling Hardware Assumptions

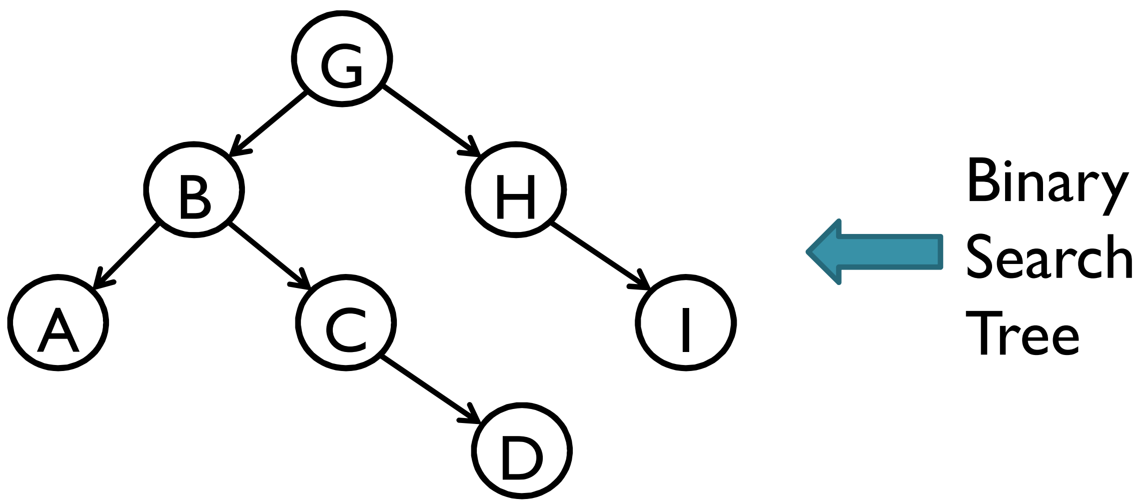
- Flash offers cheap and abundant I/O operations
  - ⇒ Can spread the DB across a log, with less physical contiguity
- Cheap high-performance data center networks
  - ⇒ Many servers can share storage, with high performance
- Large, cheap, 64-bit addressable memories
  - ⇒ Reduces the rate that Hyder needs to access the log
- Many-core web servers
  - ⇒ Hyder can afford to roll forward the log on all servers

The Hyder Stack

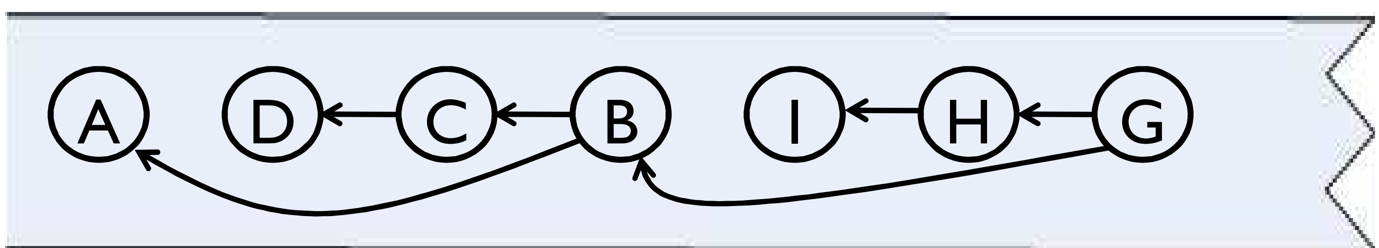


- **ISAM, SQL, LINQ, etc.**
- **Optimistic transaction protocol**
- **Multi-versioned search tree**
- **Segments, stripes and streams**
- **Append-only custom controller interface**

Database is a Search Tree

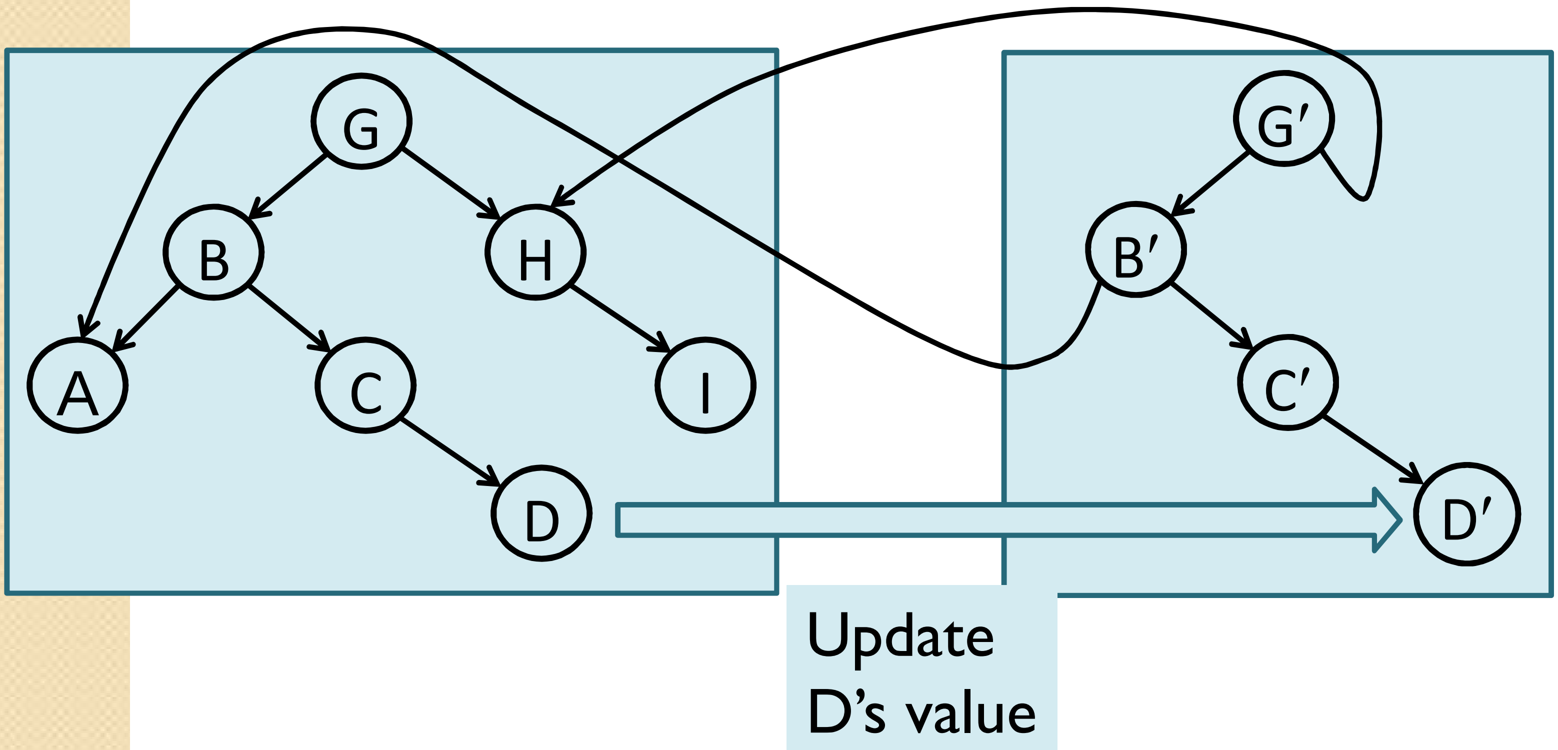


Tree is marshaled into the log

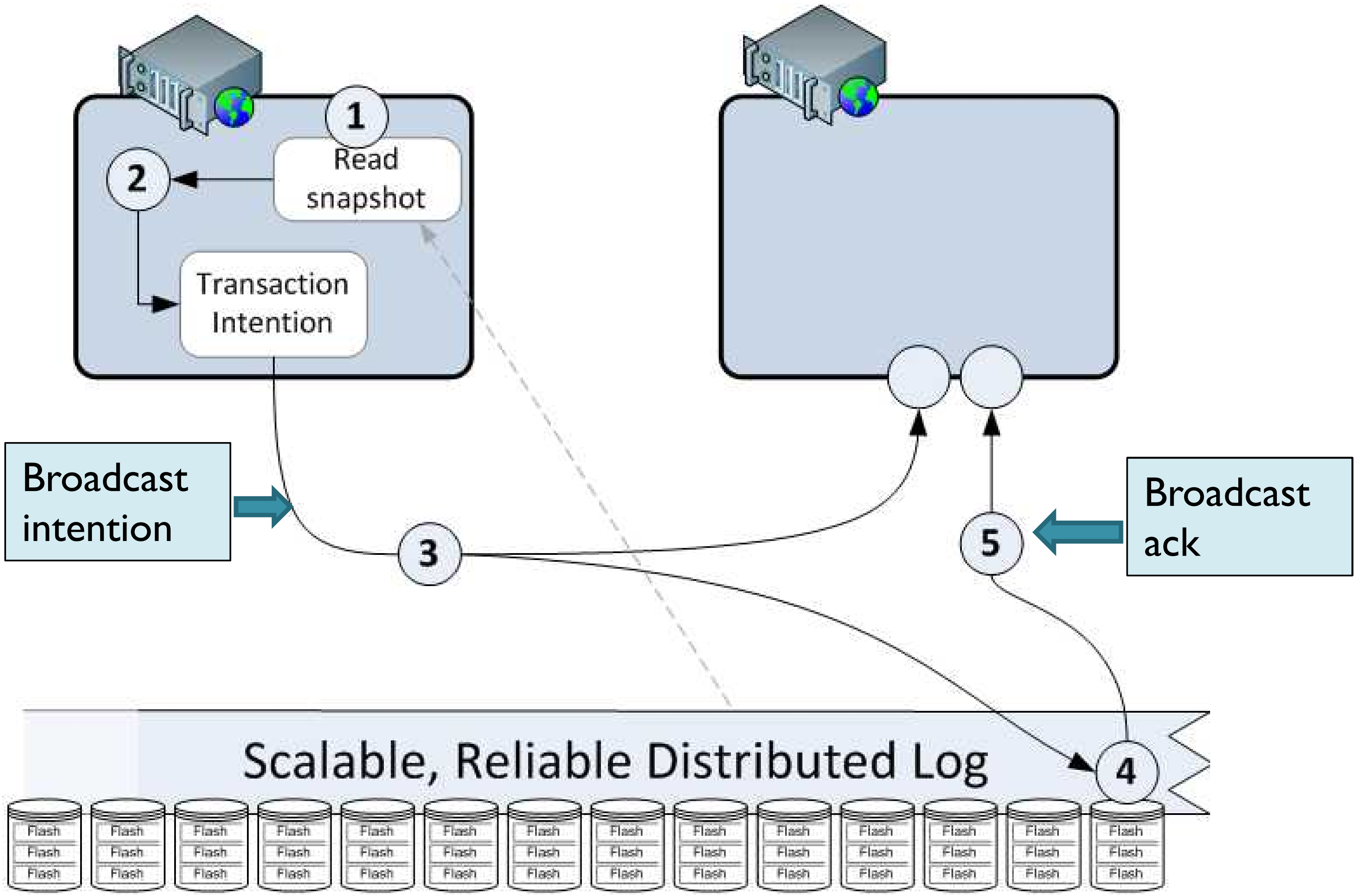


Binary Tree is Multi-versioned

- Copy on write
- To update a node, replace nodes up to the root

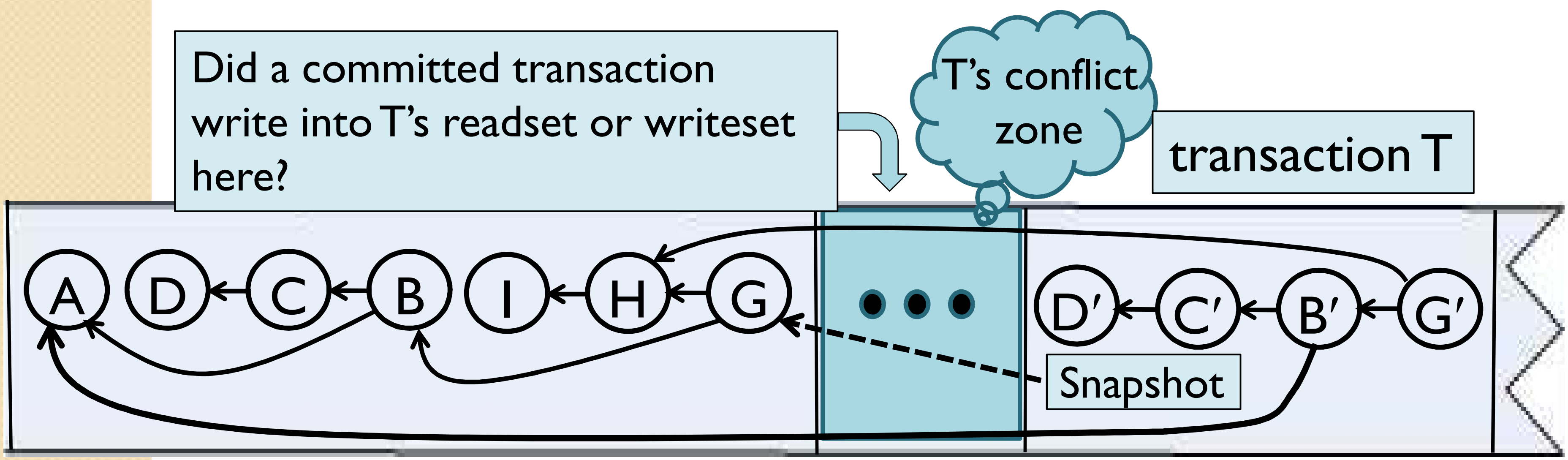


Log Updates are Broadcast



Transaction Commit

- Each server rolls forward transactions in log sequence
- When it processes an intention log record,
  - it checks whether the transaction experienced a conflict
  - if not, the transaction committed and the server merges the intention into its last committed state
- All servers make the same commit/abort decisions



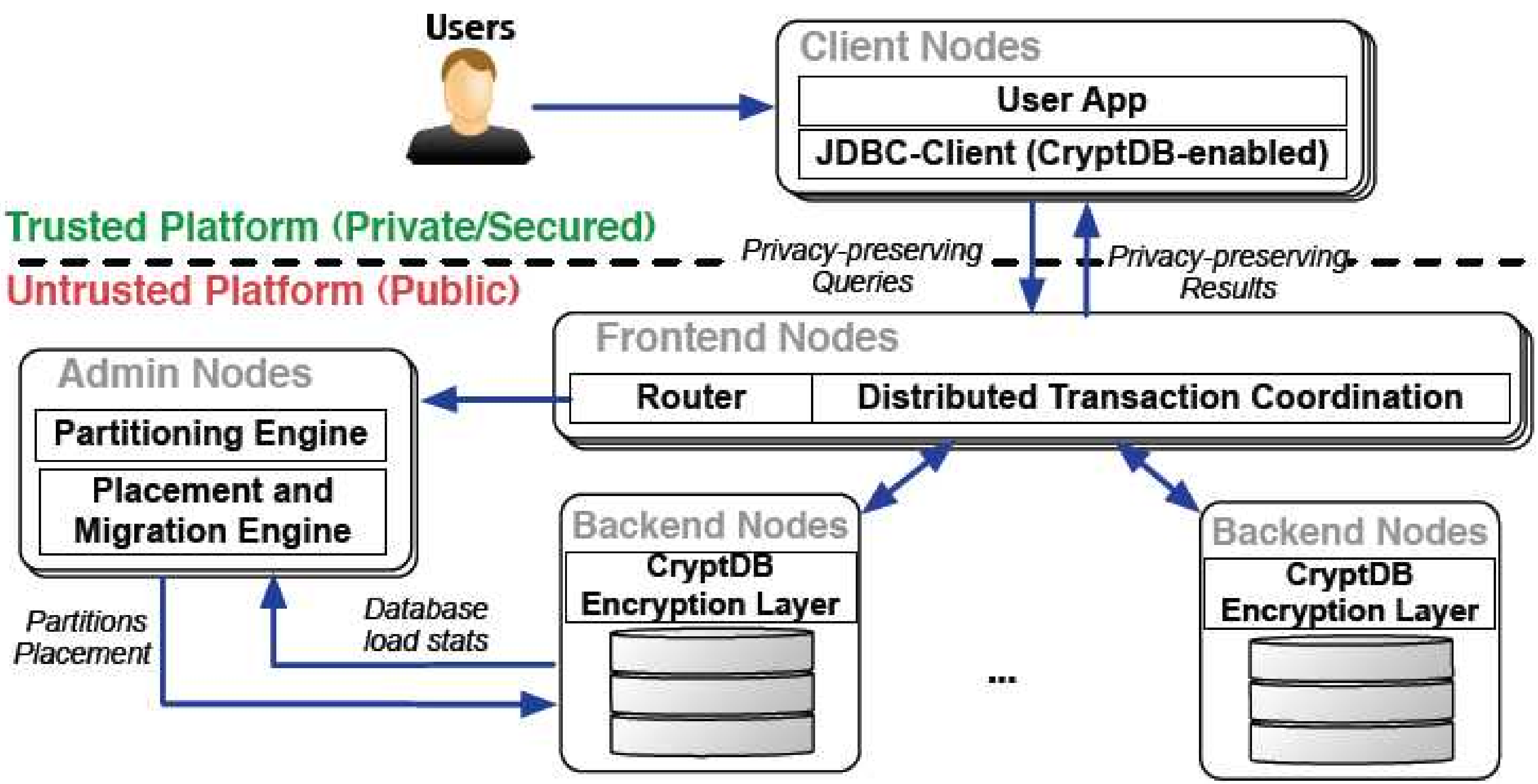
RELATIONAL CLOUD (MIT)

Relational Cloud

[Curino et al., CIDR 2011]

- Scale-out shared nothing database cluster
- Workload driven partitioning technique [Curino et al. VLDB 2010]
- Workload driven partition placement technique [Curino et al. SIGMOD 2011]

System Design



System Design

- Partition each database into one or more nodes, when the load on a database exceeds the capacity of a single machine.
- Place the database partitions on the back-end machines . Load the Database ,migrate and replicate the data for availability.
- Secure the data and process the queries.

Data Partitioning

- Two purposes:
  - to scale a single database to multiple nodes
  - to enable more granular placement.
- Relational Cloud uses a workload-aware partitioning strategy
  - Schism [discussed earlier]

Workload driven Placement

- Resource allocation is a major challenge.
- **Problems include**:
  - monitoring the resource requirements of each workload, predicting the load multiple workloads will generate when run together on a server.
- **Solution**
  - Kairos (monitoring and consolidation engine)

Workload Placement

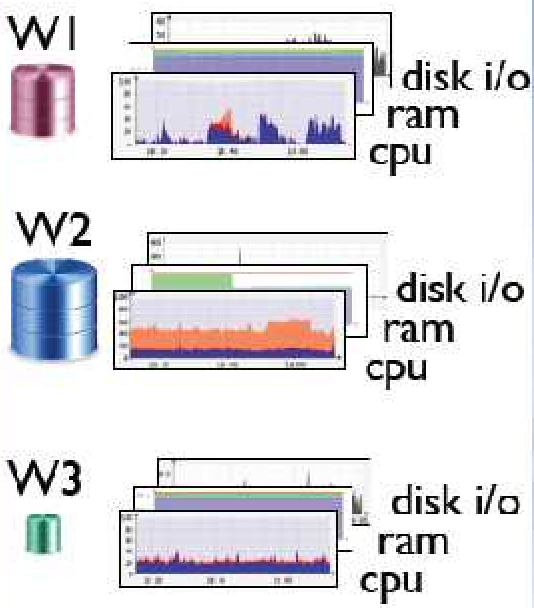
- Each workload initially run on a dedicated server
- Consolidate DB machines onto single server.

***Problem Definition*:**

- *Allocate workloads to servers in a way that:*
  - *minimizes number of servers used*
  - *balances load across servers*
  - *maintains performance unchanged*

Workload Placement

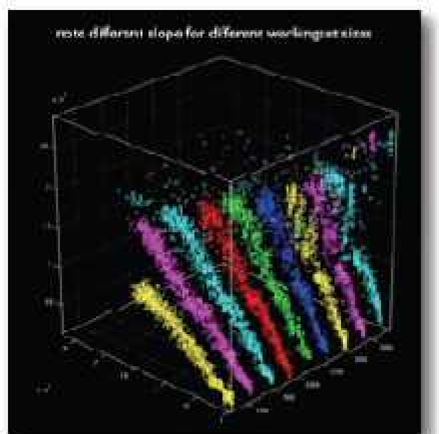
measure resource utilization



DBMSs tend to use all available resources

estimate combined load

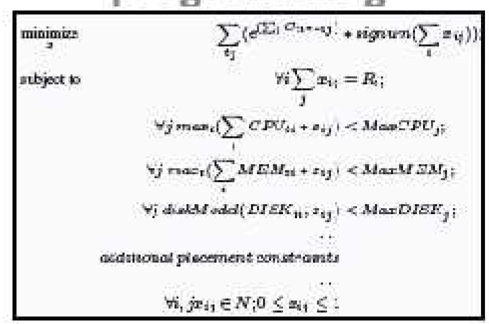
numerical models



resource non-linearities

find optimal assignment

non-linear programming



non-linear constraints and objective function

Workload Placement

Non-Linear Integer Constraints:

Problem: To determine which workloads to combine together

Goal: Minimize number of machines; maximize load balance; no resource over commitment

Input: list of machines with disk, memory, CPU, and workload profiles specifying resource utilization as (historical) time series.

servers

workloads	S1	S2	S3	S4
W1	0	0	0	1
W2	1	0	1	0
W3	1	0	0	0
W4	0	0	1	0
W5	0	0	0	1
W6	0	0	1	0
W7	1	0	0	1

Summary of Relational Cloud

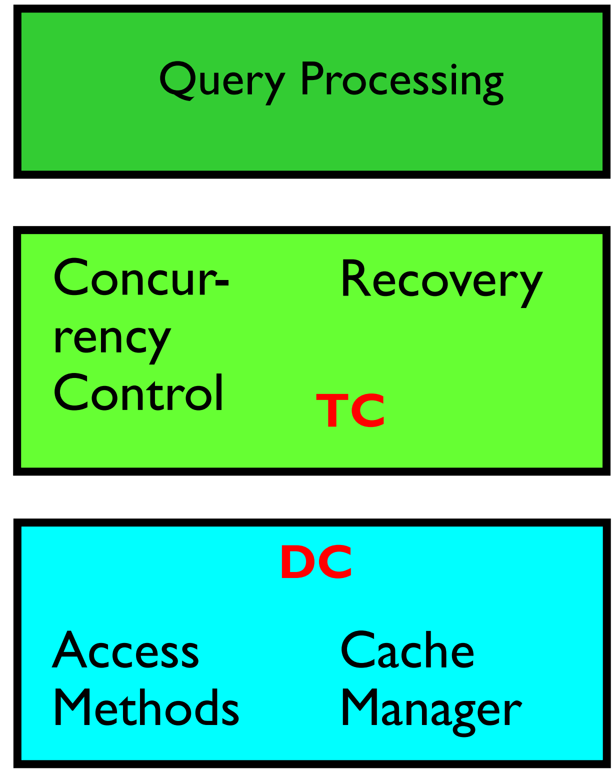
- Goals: Scalability, elasticity and privacy.
- Scalability: workload driven partitioning
  - Graph partitioning to minimize distributed transactions
- Elasticity: workload aware monitoring and consolidation
  - Optimization problem to minimize servers and maximize load balance.
- Privacy: Critical, but out of scope of this tutorial.

DEUTERONOMY (MICROSOFT)

Unbundling Transactions in the Cloud

[Lomet et al., CIDR 2009, Levandoski et al., CIDR 2011]

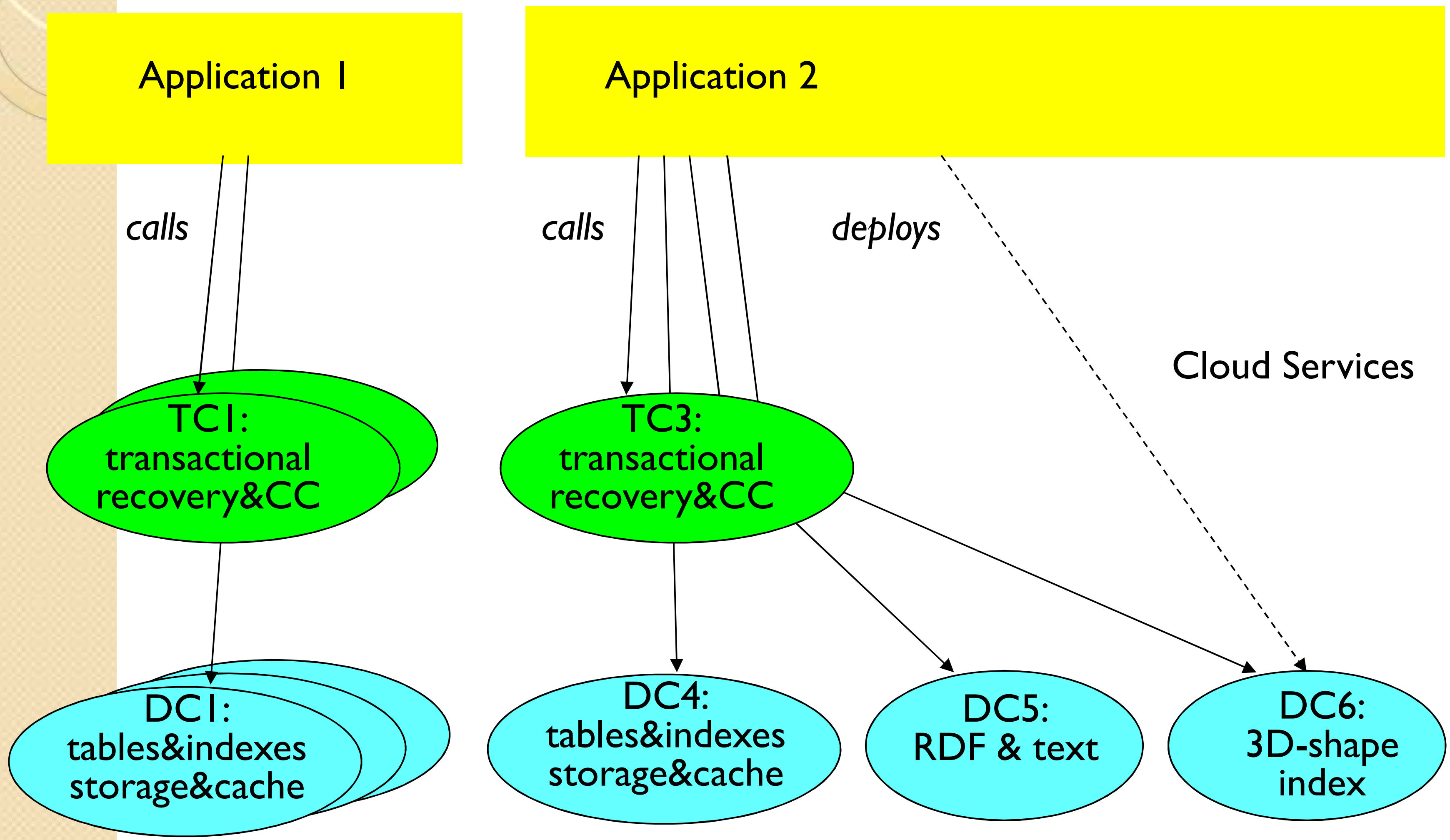
- Transaction component: TC
  - Transactional CC & Recovery
  - At logical level (records, key ranges, …)
    - No knowledge of pages, buffers, physical structure
- Data component: DC
  - Access methods & cache management
  - Provides atomic logical operations
    - Traditionally page based with latches
    - No knowledge of how they are grouped in user transactions



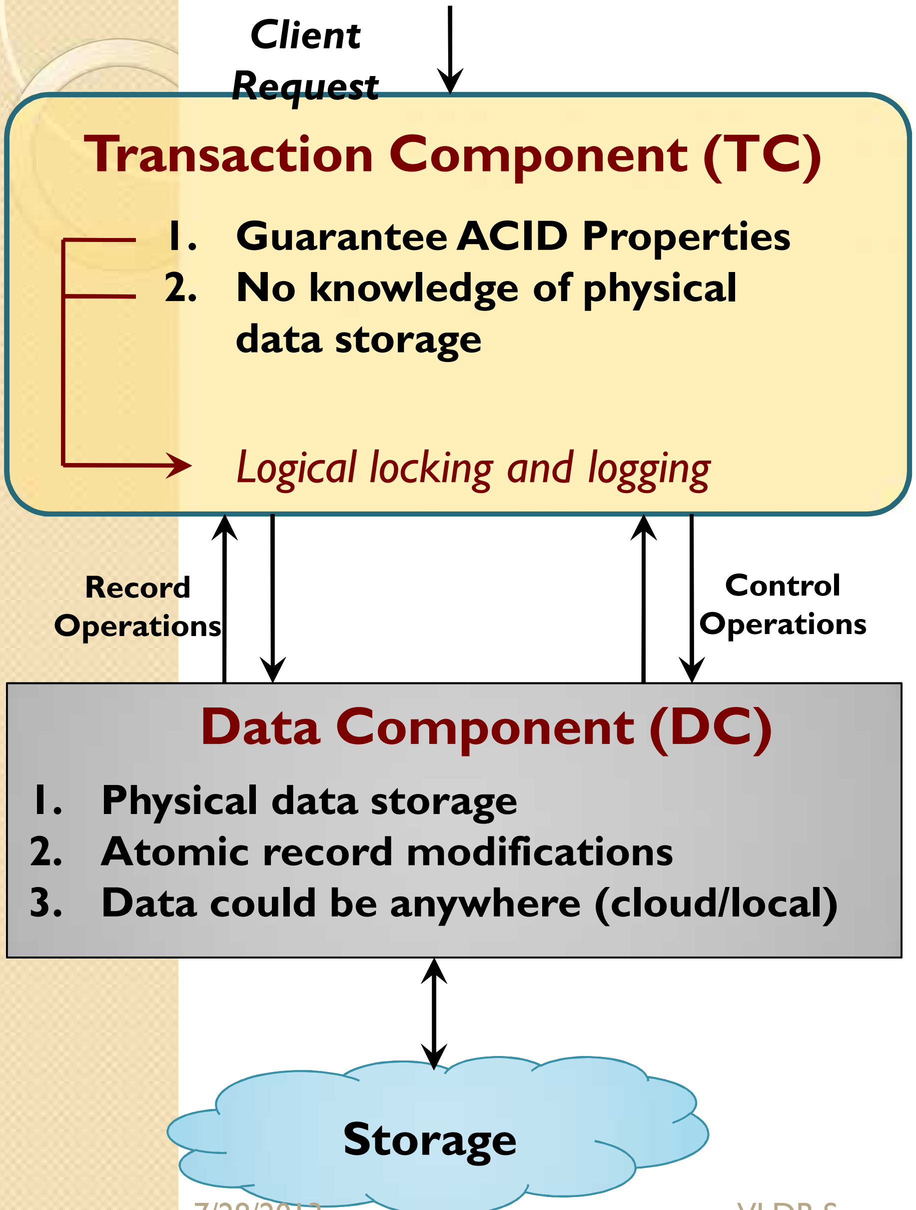
Why might this be interesting?

- Multi-Core Architectures
  - Run TC and DC on separate cores
- Extensible DBMS
  - Providing of new access method – changes only in DC
  - Architectural advantage whether this is user or system builder extension
- Cloud Data Store with Transactions
  - TC coordinates transactions across distributed collection of DCs without 2PC
  - Can add TC to data store that already supports atomic operations on data
- Major Challenge in Cloud:
  - Reduce number of round trips between TC and DC

Extensible Cloud Scenario



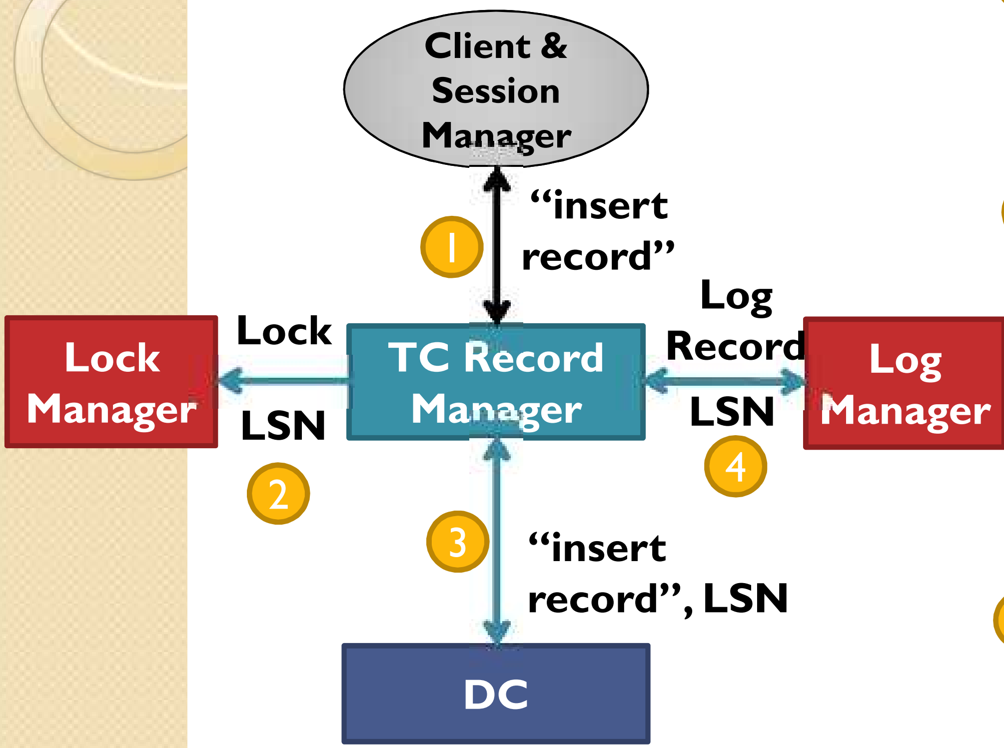
Basic Architecture



Interaction Contract

1. **Reliable messaging**
   "At least once execution"
2. **Idempotence**
   "At most once execution"
3. **Causality**
   "If DC remembers message, TC must also"
4. **Contract termination**
   "Mechanism to release contract"

Record Manager – An Insert Operation Example



1. Receive request and dispatch a session thread
2. Call to lock manager
   - Lock resource
   - Generate Log Sequence Number (LSN)
3. Sends LSN & operation to DC
4. Call to log manager
   - Log operation with LSN

Architectural Principles

- View DB kernel pieces as distributed system
- This exposes full set of TC/DC requirements
- Interaction contract (SLA) between DC & TC

And the List Continues

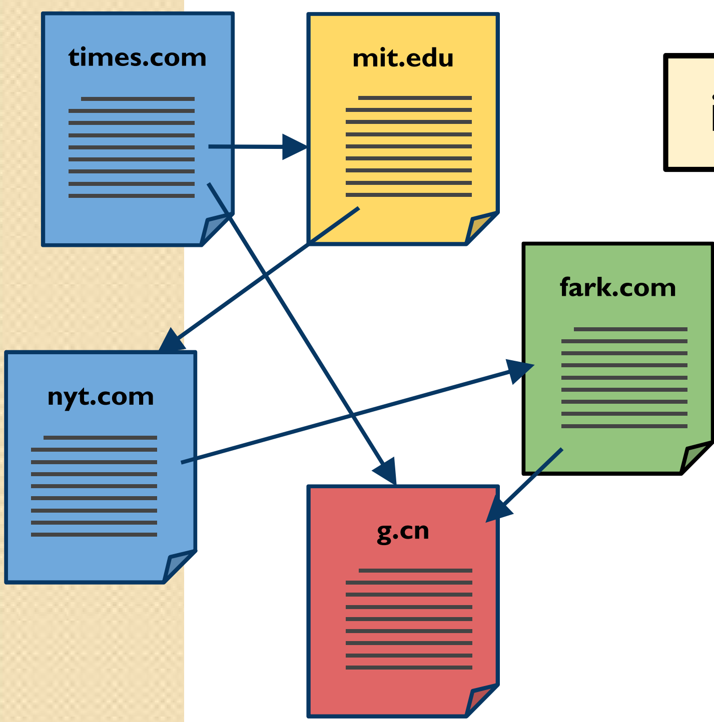
- Cloudy [ETH Zurich]
- epiC [NUS]
- Deterministic Execution [Yale]
- …

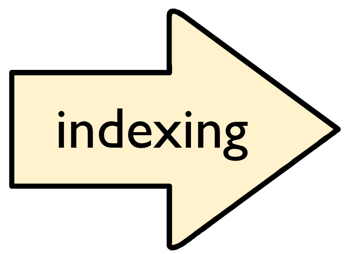
TRANSACTIONS ON DISTRIBUTED DATA: A SURVEY OF SYSTEMS

INCREMENTALLY INDEXING THE WEB WITH PERCOLATOR

Problem: Index the web

Input:
Raw documents

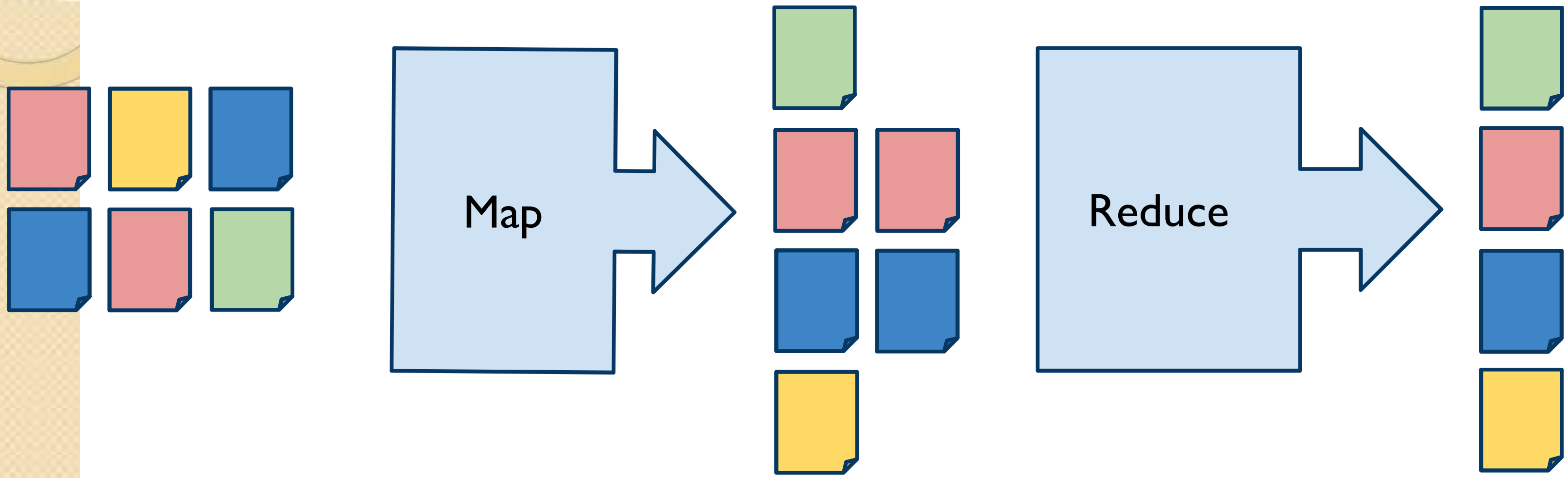




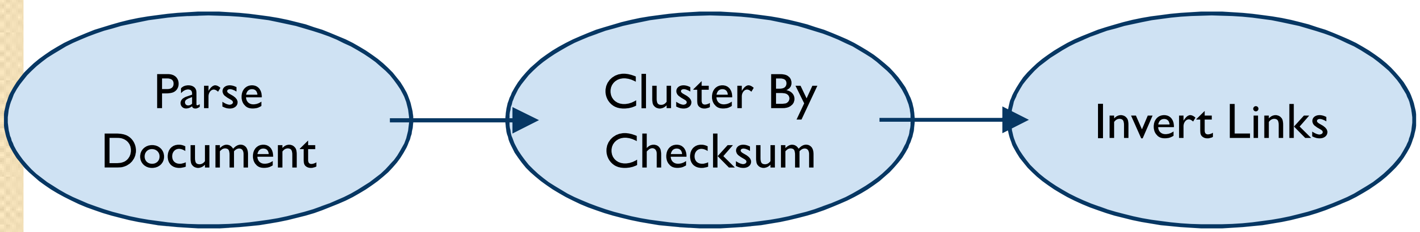
Output:
Documents ready for serving

URL	In Links	Body	PageRank
times.com	mit.edu	...	1
mit.edu	times.com	...	1
fark.com	times.com	...	3
g.cn	fark.com, times.com	....	7

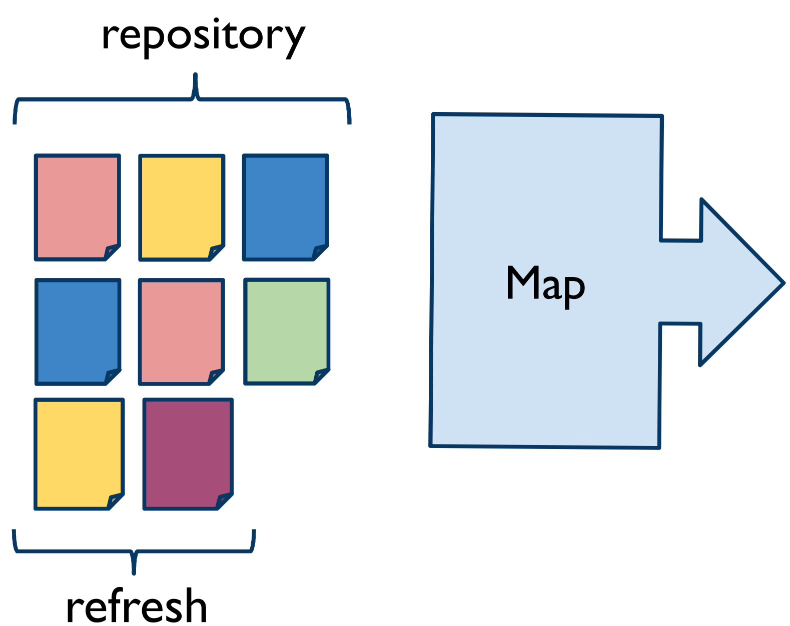
Duplicate Elimination with MapReduce



Indexing system is a chain of many MapReduces



Index Refresh with MapReduce



Should we index the new document?

- New doc could be a dup of any previously crawled
- Requires that we map over entire repository

Indexing System Goals

What do we want from an ideal indexing system?

- Large repository of documents
  - Upper bound on index size
  - Higher-quality index: e.g. more links
- Small delay between crawl and index: "freshness"

MapReduce indexing system: Days from crawl to index

Incremental Indexing

- Maintain a random-access repository in Bigtable
- Indices let us avoid a global scan
- Incrementally mutate state as URLs are crawled

URL	Contents	Pagerank	Checksum	Language
http://usenix.org/osdi10	<html>CFP, ...	6	0xabcdef01	ENGLISH
http://nyt.com/	<html>Lede ...	9	0xbeefcafe	ENGLISH

Incremental Indexing on Bigtable

URL	Checksum	PageRank	IsCanonical?
nyt.com	0xabcdef01	6	yeno
nytimes.com	0xabcdef01	9	yes

Checksum	Canonical
0xabcdef01	nnytimes.com

What happens if we process both URLs simultaneously?

Percolator: Incremental Infrastructure

Adds distributed transactions to Bigtable

```
(0) Transaction t;
(1) string contents = t.Get(row, "raw", "doc");
(2) Hash h = Hash32(contents);
    ...
    // Potential conflict with concurrent execution
(3) t.Set(h, "canonical", "dup_table", row);
    ...
(4) t.Commit();  // TODO: add retry logic
```

Simple API: Get(), Set(), Commit(), Iterate

Implementing Distributed Transactions

- Provides *snapshot isolation* semantics
- Multi-version protocol (mapped to Bigtable timestamps)
- Two phase commit, coordinated by client
- Locks stored in special Bigtable columns:

"balance"

	balance:data	balance:commit	balance:lock
Alice	5: 4: 3: $10	5: 4: data @ 3 3:	5: 4: 3:

Transaction Commit

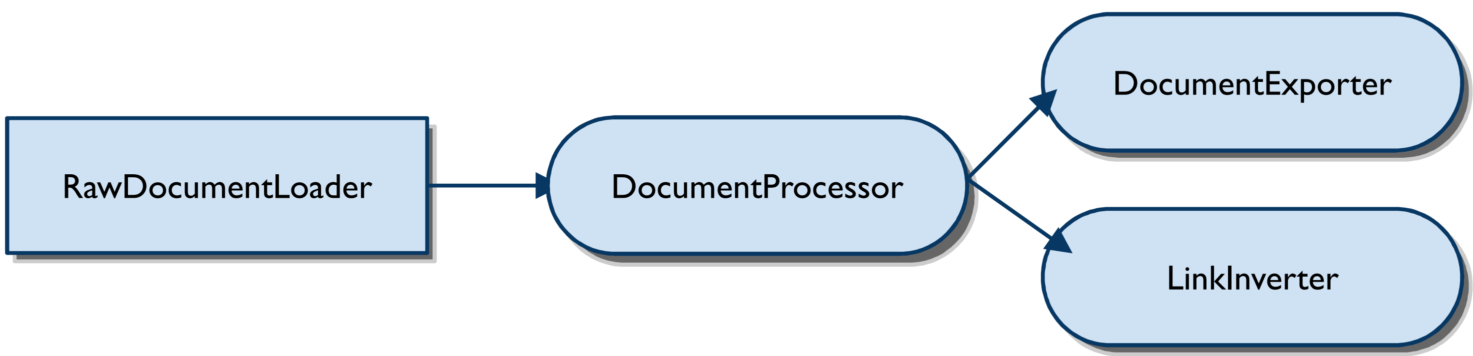
```
Transaction t;
int a_bal = t.Get("Alice", "balance");
int b_bal = t.Get("Bob", "balance");
t.Set("Alice", "balance", a_bal + 5);
t.Set("Bob", "balance", b_bal - 5);
t.Commit();
```

	balance:data	balance:commit	balance:lock
Alice	5: $15 4: 3: $10	6: data @ 5 5: 4: data @ 3 3:	5: 4: 3:
Ben	5: $5 4: 3: $10	6: data @ 5 5: 4: data @ 3 3:	5: 4: 3:

Notifications: tracking work

Users register "observers" on a column:
- Executed when any row in that column is written
- Each observer runs in a new transaction
- Run at most once per write: "message collapsing"

Applications are structured as a series of Observers:



Implementing Notifications

Dirty column: set if observers must be run in that row

Randomized distributed scan:

- Finds pending work, runs observers in thread pool
- Scan is efficient: only scans over bits themselves

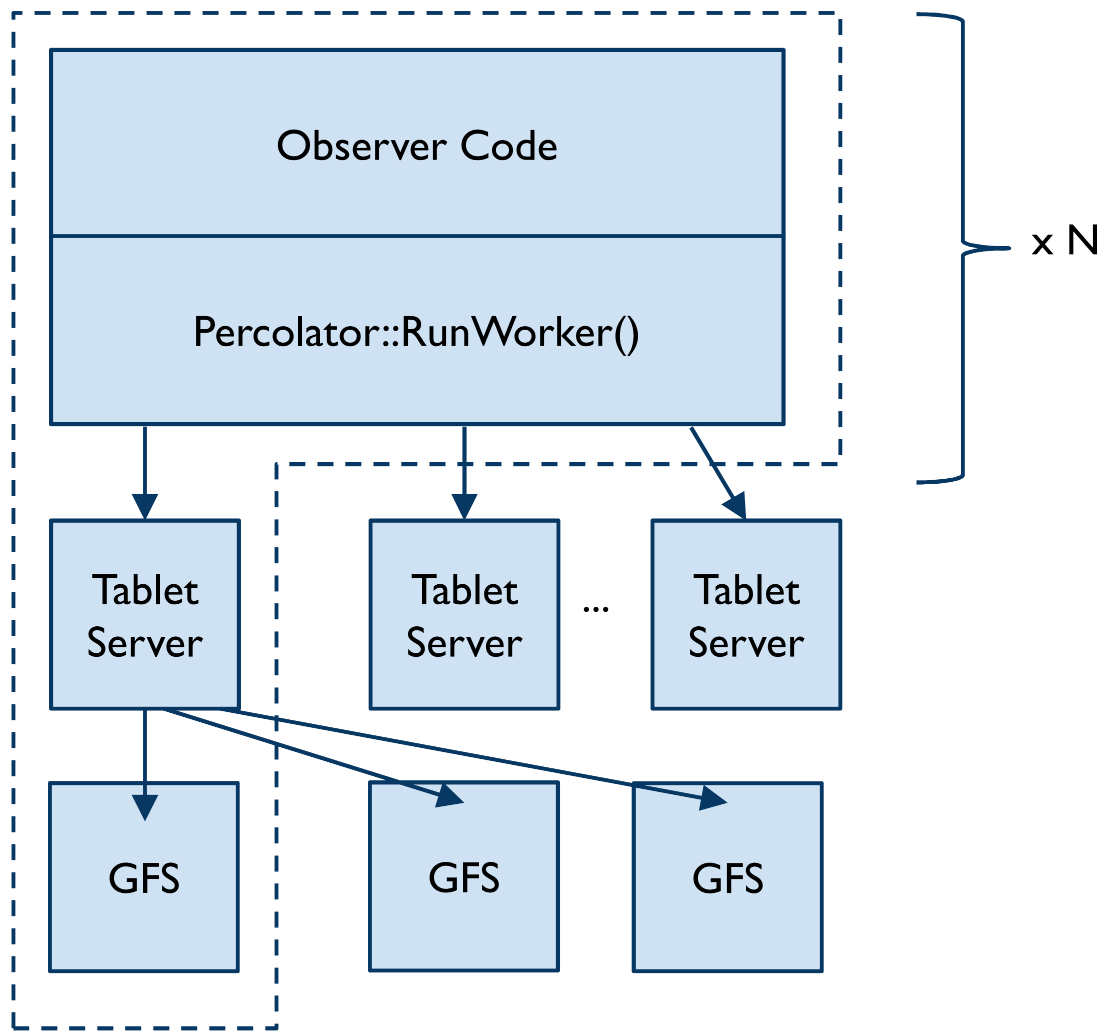
No shards or work units: nothing to straggle

	Dirty?	**balance:data**	**...**
Alice	Yes	5: $15	
Bob	No	5: $5	

Running Percolator

Each machine runs:

- Worker binary linked with observer code.
- Bigtable tablet server
- GFS chunkserver

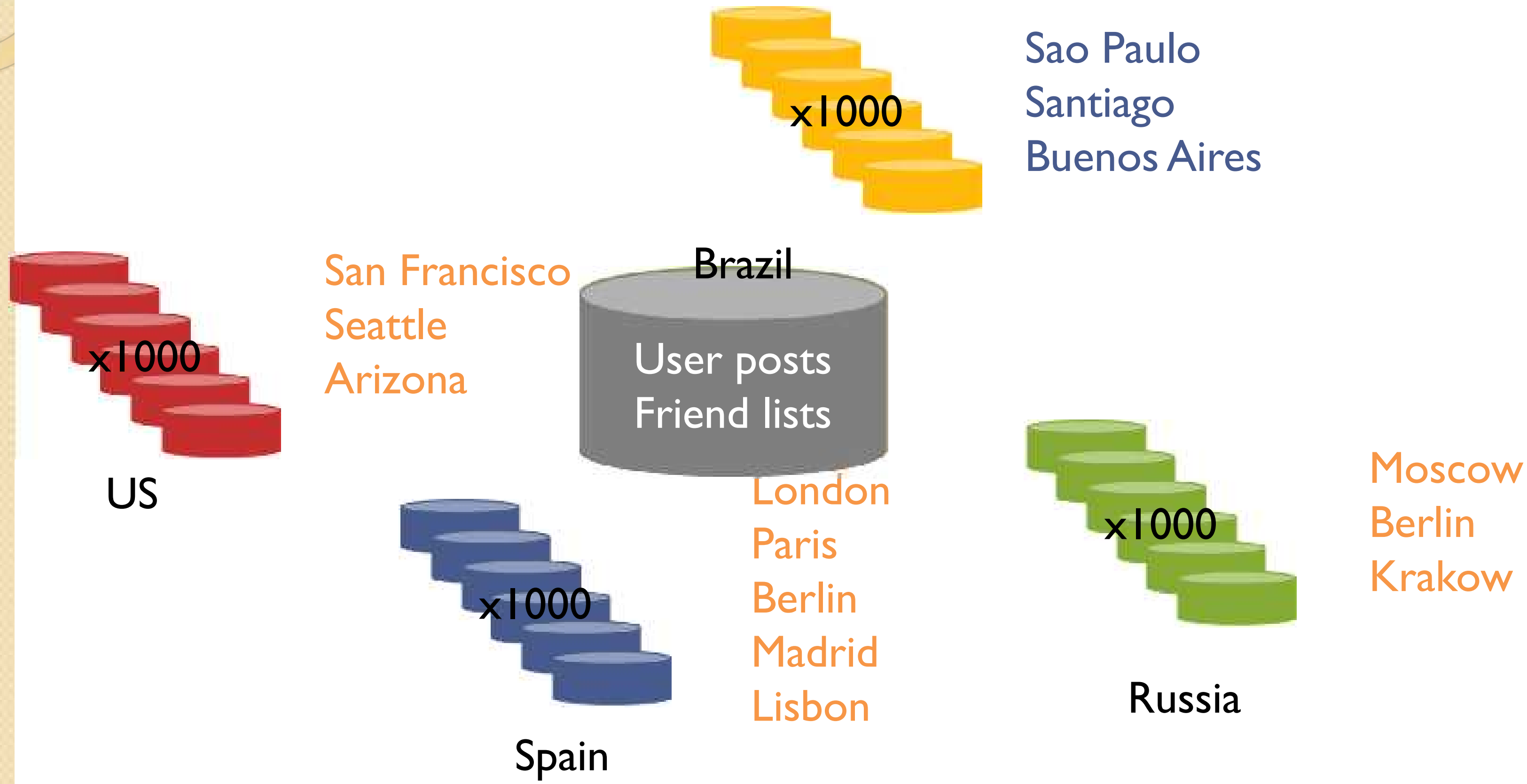


SPANNER

What is Spanner?

- Distributed multiversion database
  - General-purpose transactions (ACID)
  - SQL query language
  - Schematized tables
  - Semi-relational data model
- Running in production
  - Storage for Google's ad data
  - Replaced a sharded MySQL database

Example: Social Network



Overview

- Feature: Lock-free distributed read transactions
- Property: External consistency of distributed transactions
  - First system at global scale
- Implementation: Integration of concurrency control, replication, and 2PC
  - Correctness and performance
- Enabling technology: TrueTime
  - Interval-based global time

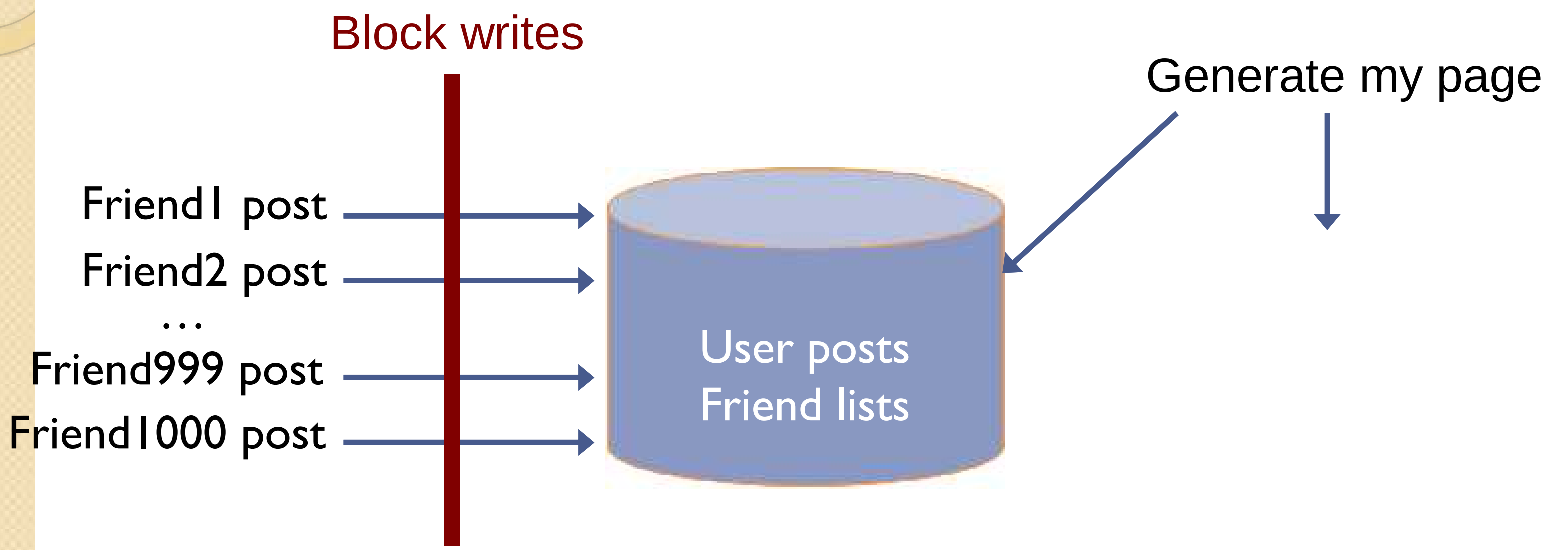
Read Transactions

- Generate a page of friends' recent posts
  - Consistent view of friend list and their posts

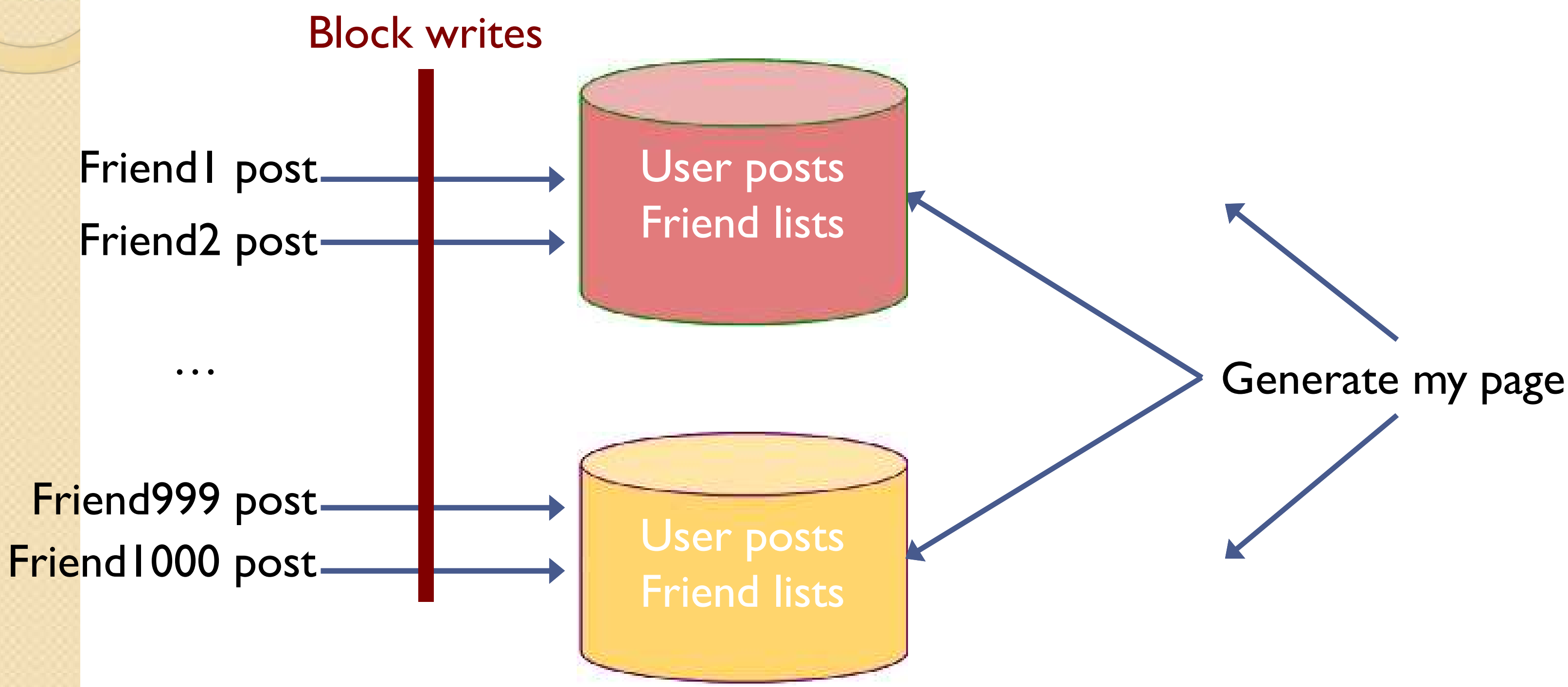
Why consistency matters

1. Remove untrustworthy person X as friend
2. Post P: "My government is repressive…"

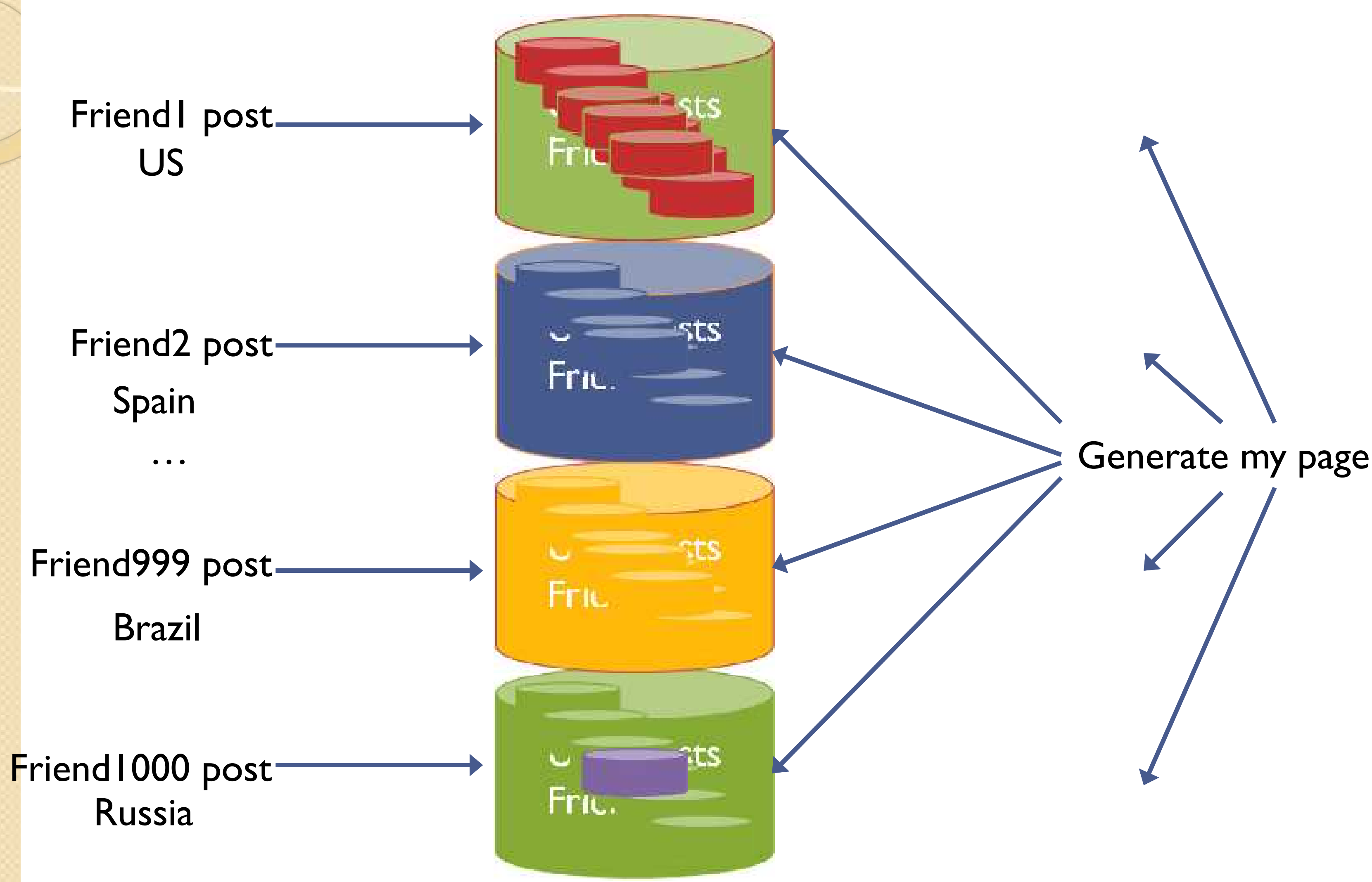
Single Machine



Multiple Machines



Multiple Datacenters



Version Management

- Transactions that write use strict 2PL
  - Each transaction *T* is assigned a timestamp *s*
  - Data written by *T* is timestamped with *s*

Time	<8	8	15
My friends	[X]	[]	
My posts			[P]
X's friends	[me]	[]	

Synchronizing Snapshots

Global wall-clock time

==

External Consistency:

Commit order respects global wall-time order

==

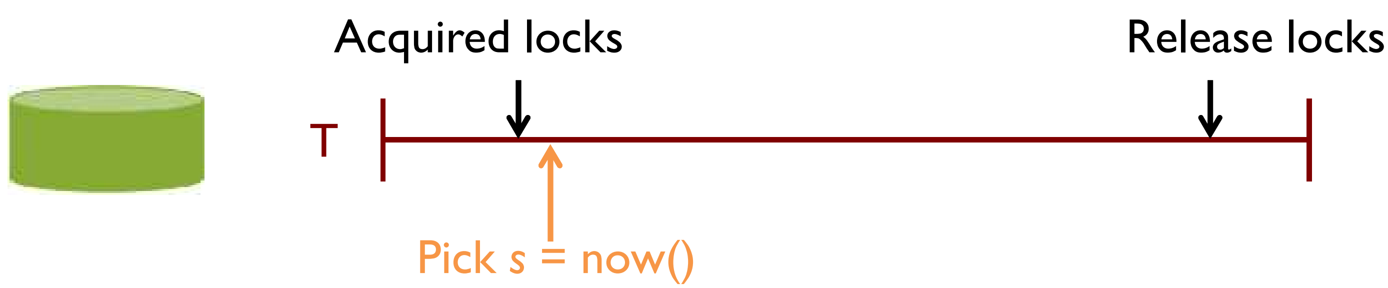
Timestamp order respects global wall-time order

given

timestamp order == commit order

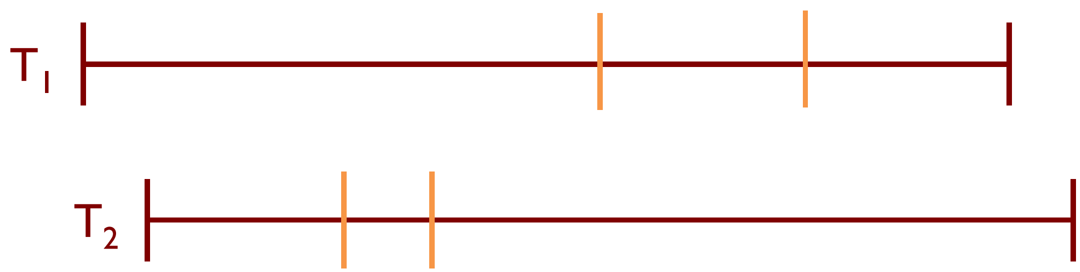
Timestamps, Global Clock

- Strict two-phase locking for write transactions
- Assign timestamp while locks are held

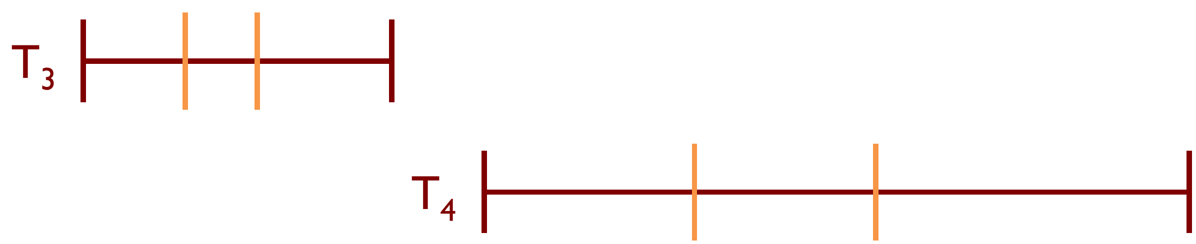


Timestamp Invariants

- Timestamp order == commit order

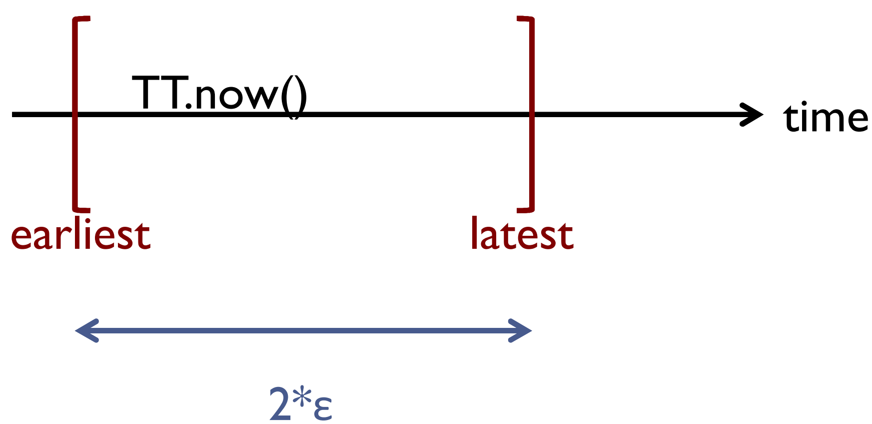


- Timestamp order respects global wall-time order

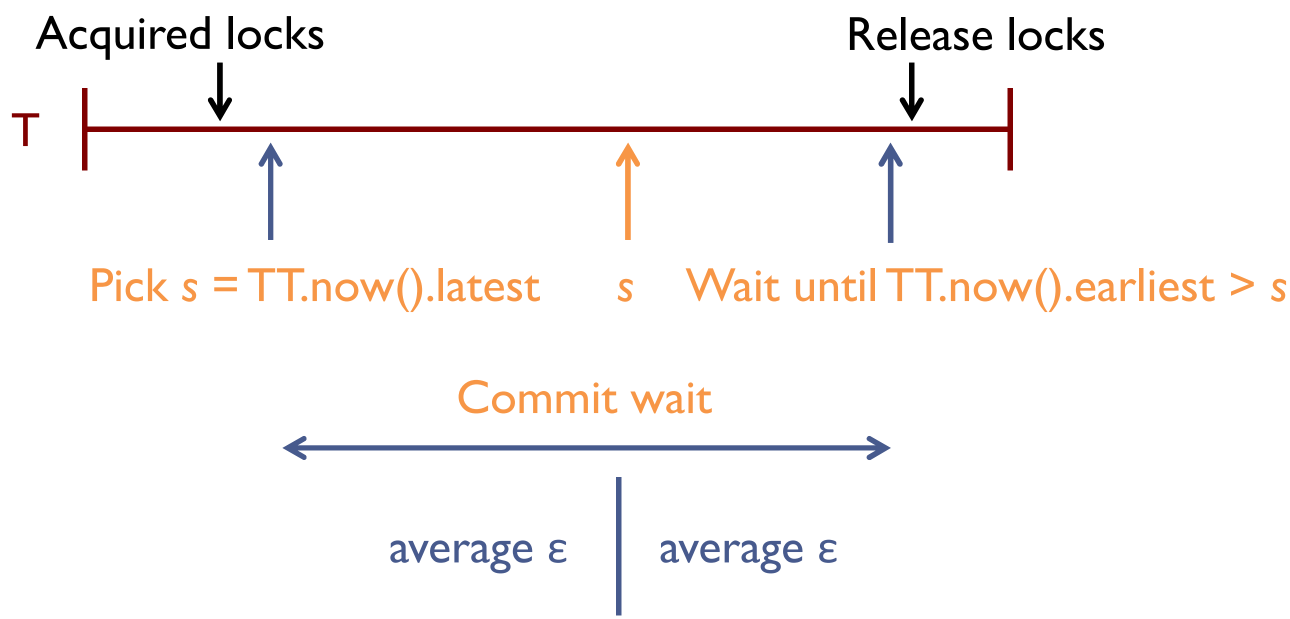


TrueTime

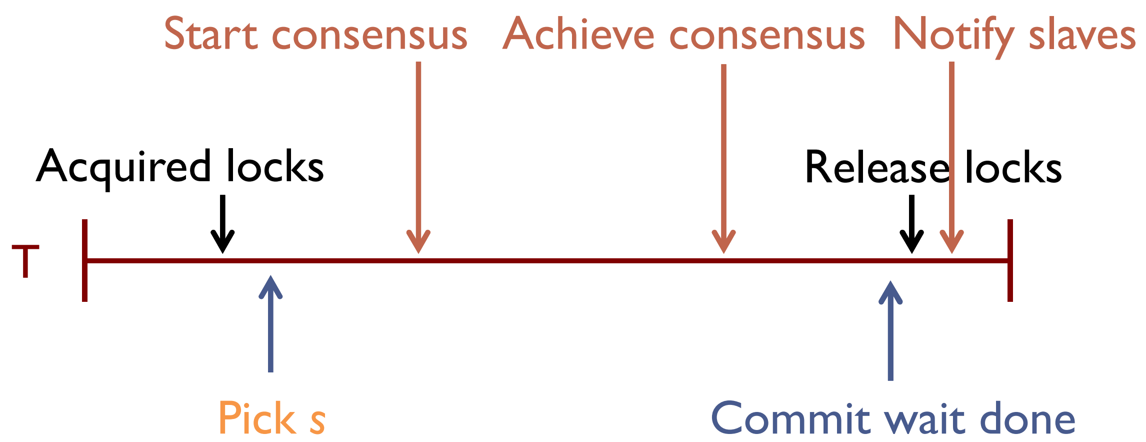
- "Global wall-clock time" with bounded uncertainty



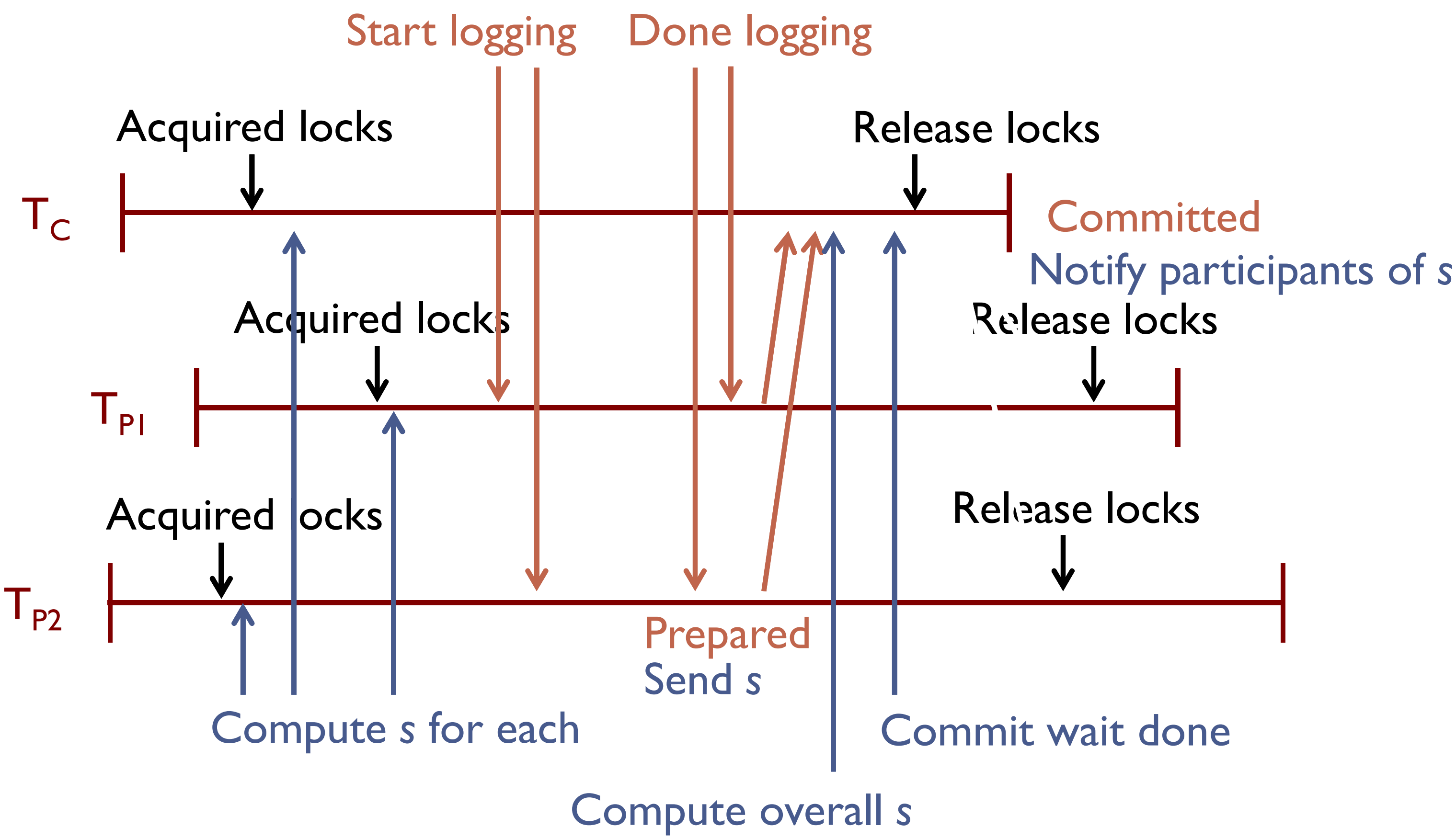
Timestamps and TrueTime



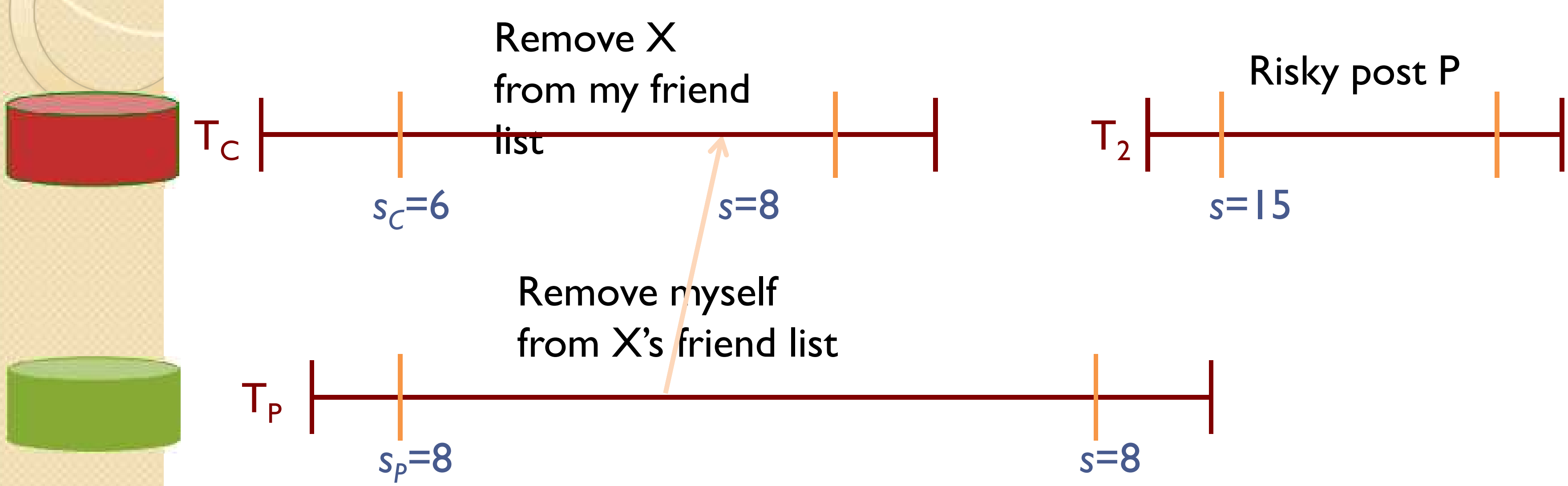
Commit Wait and Replication



Commit Wait and 2-Phase Commit

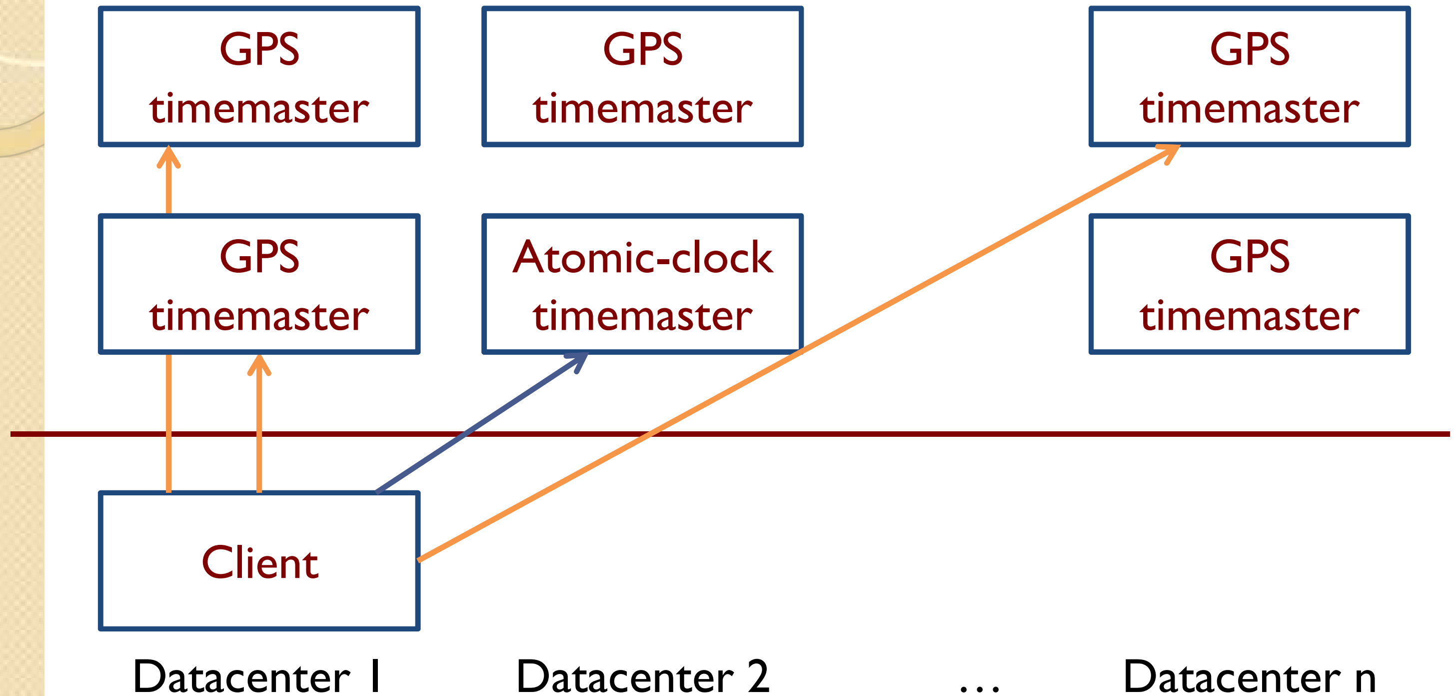


Example



Time	<8	8	15
My friends	[X]	[]	
My posts			[P]
X's friends	[me]	[]	

TrueTime Architecture

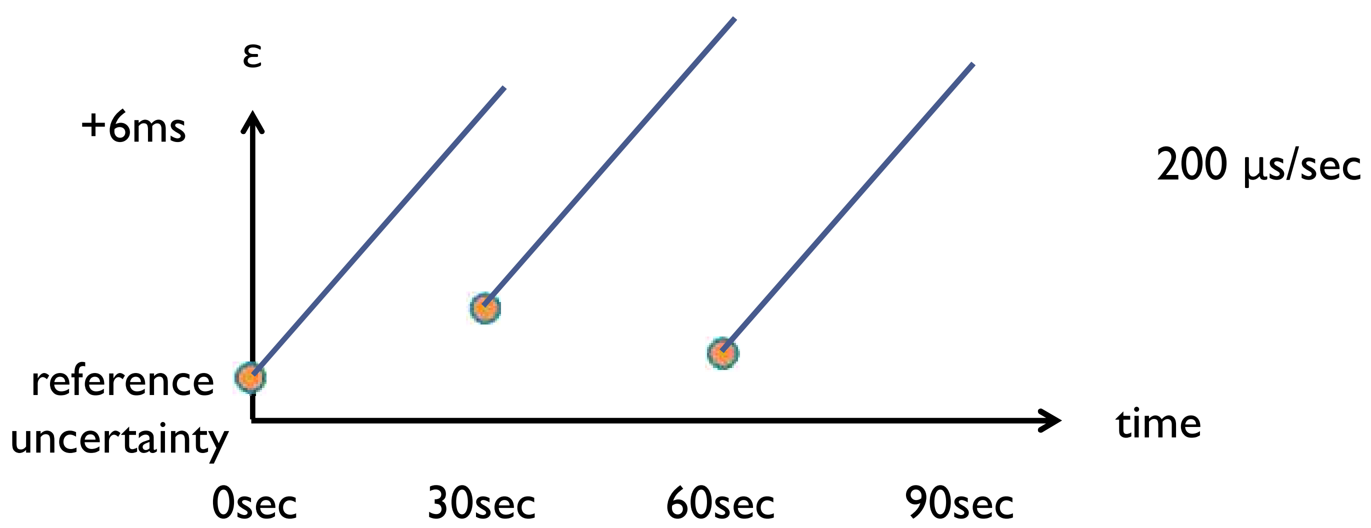


Compute reference [earliest, latest] = now ± ε

TrueTime implementation

now = reference now + local-clock offset

ε = reference ε + worst-case local-clock drift



What If a Clock Goes Rogue?

- Timestamp assignment would violate external consistency
- Empirically unlikely based on 1 year of data
  - Bad CPUs 6 times more likely than bad clocks

Discussion

- Transactional guarantees on distributed data
  - Distributed synchronization is inevitable
- We discussed a few production systems that explore different points of the space
- The exact system of choice is often dependent on the application’s requirements